DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/91D-26

Parallelizing Serial Code in a Distributed Processing Environment with an Application in High Frequency Electromagnetic Scattering

THESIS

Paul R Work
2 Lt, USAF

AFIT/GCS/ENG/91D-26

Approved for public release; distribution unlimited

AFIT/GCS/ENG/91D-26

Parallelizing Serial Code for a Distributed Processing Environment with an Application to High Frequency Electromagnetic Scattering

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Paul R Work, B.S.
2 Lt, USAF

December, 1991

Acknowledgments

First and foremost, I would like to thank my wonderful wife, Sherry, who has patiently supported me through the difficult times of the last year. She has not only encouraged me when frustrations arose, but also served as a proof reader. In that role, she has suggested many changes to improve the readability of this thesis. I would also like to thank Rick Norris for his help in learning the idiosynchracies of the iPSC/2 and iPSC/860 hypercubes. His expert knowledge often saved me hours of frustration as glitches and bugs appeared at regular intervals. Another person worthy of mention is Tony Schooler, whose in depth knowledge of LaTeX has proven invaluable in applying some of the more complex functions available. I would also like to thank Shala Arshi and Wendy Wilhelm at Intel for providing valuable expert assistance with and access to their 64 node cube. Finally, I would like to thank Dr. Gary B. Lamont for his encouragement, gentle proddings, and support throughout this thesis effort.

Paul R Work

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/91D-26

Abstract

This research investigates the parallelization of existing serial programs in computational electromagnetics for use in a parallel environment. Existing algorithms for calculating the radar cross-section of an object are covered, and a ray-tracing code is chosen for implementation on a parallel machine. Current parallel architectures are introduced and a suitable parallel machine is selected for the implementation of the chosen ray tracing algorithm. The standard techniques for the parallelization of serial code are discussed, including load balancing and decomposition considerations,and appropriate methods for the parallelization effort are selected. A load balancing algorithm is modified to increase the efficiency of the application, and a high level design of the structure of the serial program is presented. A detailed design of the modifications for the parallel implementation is also included, with both the high level and the detailed design specified in a high level design language called UNITY. The correctness of the design is proven using UNITY and standard logic operations. The theoretical and empirical results show that it is possible to achieve an efficient parallel application of a serial computational electromagnetic program where the characteristics of the algorithm and the target architecture critically influence the development of such an implementation.

Parallelizing Serial Code for a Distributed Processing Environment with an Application to High Frequency Electromagnetic Scattering

I. Problem Description

1.1 Background

This chapter introduces the basic concepts that are used in this thesis investigation. Radar and its uses are discussed, and some of the methods for simulating the effects of radar are introduced: ray tracing and matrix modeling. The general thrust of this research is explained and a quick summary of the current knowledge is presented.

1.1.1 Fundamentals of Radar Shortly after World War II began, British scientists developed a method for tracking flying aircraft using electromagnetic waves. This method was called radar which stands for *ra*dio *d*etection *a*nd *r*anging (16). Later, additional abilities were added including the tracking of ships, land based vehicles, and even terrain mapping and avoidance. This tracking ability was refined to the point where the data generated from a high precision radar set could be used to guide a missile or an anti-aircraft battery to destroy the target. Radar helps both friend and foe to follow the movements of airplanes, ships, and ground vehicles. It can also aid in the destruction of a target. The ease with which a radar site tracks an object is directly related to the radar cross-section (RCS) of that object. Military agencies and manufacturers are therefore concerned about the RCS of the items they construct and maintain.

A radar cross-section is a pattern of reflected and diffracted electromagnetic (EM) waves which emanate from a given area or object that is illuminated by a transmitting antenna. Figure 1 shows that when radar waves from a transmitter encounter a target, these waves scatter in all directions. The object or area that scatters the incident energy is called the "scene", and a scene is said to be illuminated when energy from a transmitting antenna encounters that scene. When incident EM waves strike a scene, some of these waves are reflected by parts of the scene geometry, while others are diffracted by other details in

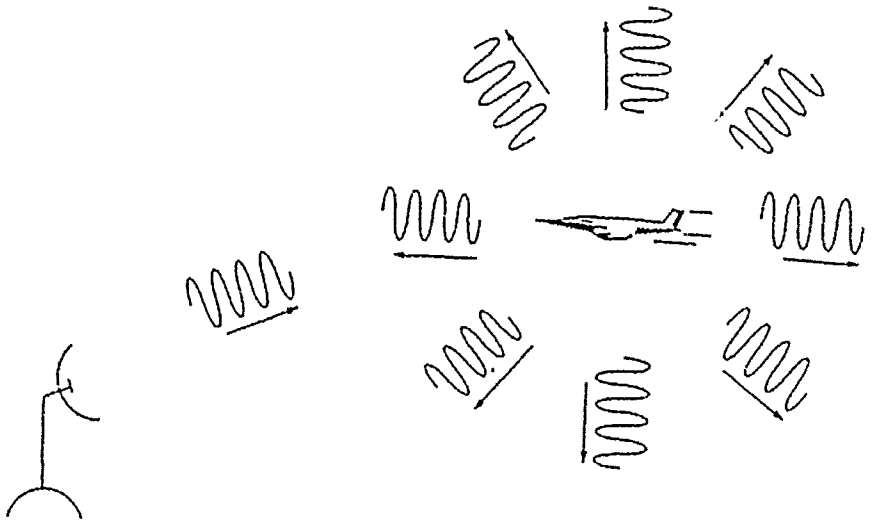

Figure 1. Reflecting electromagnetic energy off an object

the scene. The incident EM waves can also undergo both reflection and diffraction before leaving the area. Some of the reflected and diffracted radar waves may be detected by a receiving antenna and analyzed for information about the illuminated scene. A receiving antenna may be located at any angle away from the scene relative to the transmitter, so the response of the scene at all angles is necessary when analyzing a design or existing object. The returned EM energy that is picked up by a receiving antenna is only a part of the total pattern of scattered EM energy associated with an RCS and is known as a radar echo. Figure 2 shows an example of an RCS for a B-26 aircraft of World War 2 vintage. This figure shows the relative power that a receiving antenna would detect if it were to be placed at the azimuth angle indicated in a polar coordinate system relative to the object itself.

1.1.2 Calculating a Radar Cross Section Early in the history of electronic computers, researchers proposed using them in the field of image synthesis to create a simulated view of an object or set of objects called a scene (11). This approach was called "ray-tracing". Unfortunately, the techniques designed for this purpose were computationally intensive, and the computers of that day were not powerful enough to solve the problems in a reasonable amount of time. For this reason, little work was done in this field until the 1980's. Most work in image synthesis has been in the realm of optical renderings of a scene from a single viewpoint. By changing the frequency of the incident energy, ray-tracing can

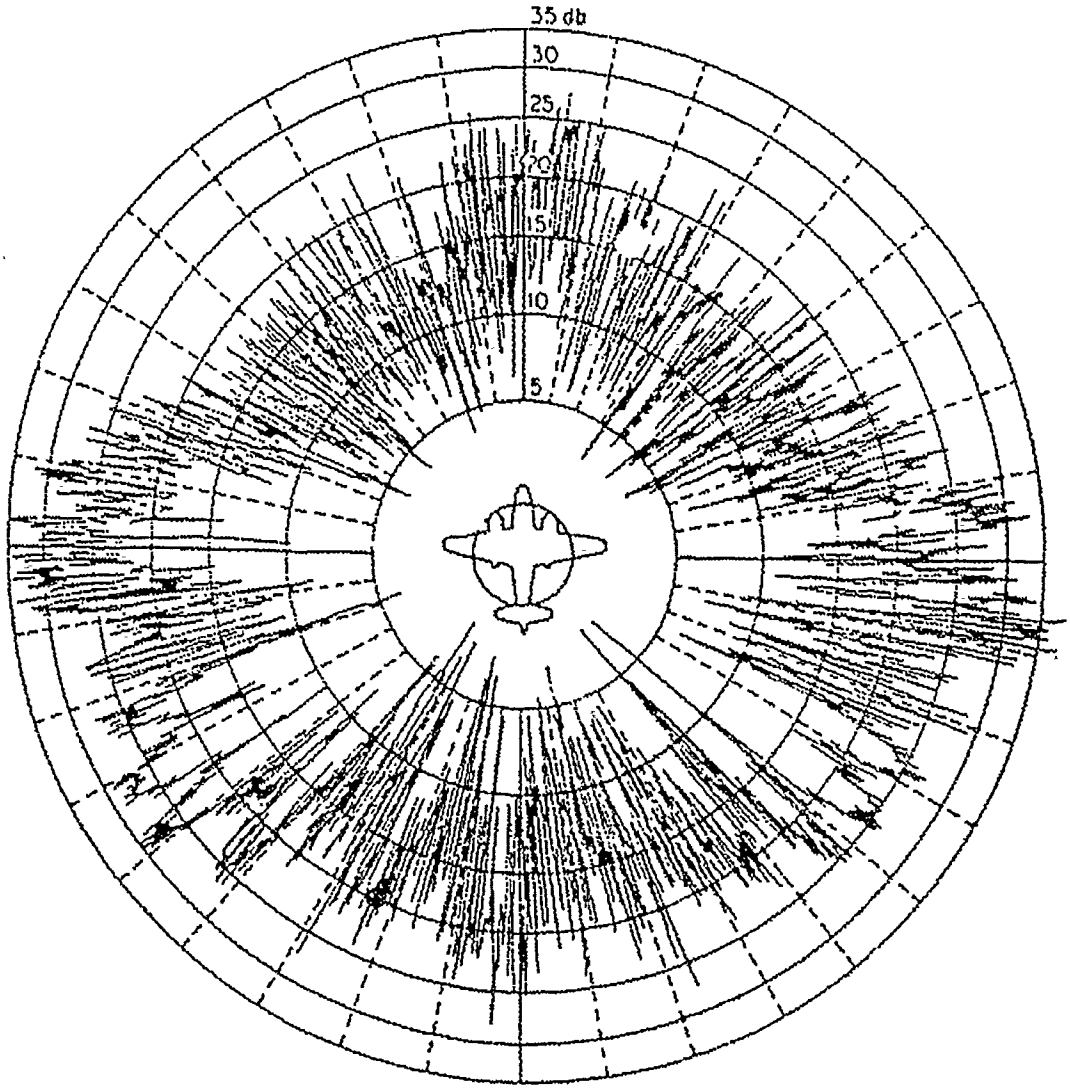

Figure 2. Experimentally measured RCS of the B-26 as a function of azimuth angle

be used to generate an electromagnetic image of a scene. By changing the viewpoint from one location to multiple locations, a simulated radar cross section can be computed.

Another method for calculating the EM fields that result from an interaction between an object and an incident field uses vector matrix models to calculate the initial surface currents that are generated by incident EM waves. These current values are then transformed into another set of matrices, and these matrices are used to calculate a resultant EM field based on linear equation structures.

The United States Air Force is currently conducting research into methods which would permit computers to more efficiently calculate the radar cross-section of a complex object. The application of this research is supported by two different areas of EM scattering simulation. The first area is concerned with the radar observability of friendly aircraft and vehicles, and the other area in EM scattering simulation lies in the realm of target recognition. Researchers in the first area are concerned with the RCS of their own designs. When planning new designs or modifications to an existing vehicle, they want to know the radar cross-section of the resulting design before going into production. Efforts in the second area seek to identify possibly hostile targets based in part on empirical data.

1.1.3 Techniques for finding an RCS One method for determining the RCS of an object involves building a scale model of a proposed design. This scale model can then be placed in a simulated environment, and measurements can be taken at all angles as EM waves are transmitted at the model. Measurements can also be made using a full-size example of a vehicle on an electromagnetic range or in the field. Models take time to build and are not always an accurate reproduction of the final design. Field measurements, on the other hand, can be expensive or extremely difficult to achieve with accurate results. Both of these methods require some time to gather the necessary data. Another way to generate an RCS is to calculate the EM field that would result if an incident EM field of known intensity were to encounter an object. With an accurate mathematical model of the object, the theoretical result can be calculated. This becomes the method of choice when field measurements are costly, difficult, or impossible, and computational facilities are effective and efficient.

In the 1980's, advancements in the field of computer architecture resulted in the creation of *parallel* computers (10). These newer computers have many processors linked together. Traditional computers have only one central processing unit and have the speed of light as a physical limitation on their processing power (11). This limitation arises from

the fact that an electrical impulse travels at the speed of light, and all transactions within a computer take the form of electrical signals transmitted on wires or traces. If the speed of transmission is limited by the speed of light, then a limited number of operations can take place within a given period of time given that different components are physically separate from one another. Although an individual processor in a parallel machine usually has less processing power than a CPU of a large-scale traditional computer, because a parallel machine can have many of these processors, the total computing power of the parallel computer can exceed that of the traditional computer by a large margin. However, because of the recent advent of parallel computers and the difficulty of the conversion process, relatively few programs have been converted to run on these architectures.

Today, there are many specific algorithms for calculating the EM field that results when an object or group of objects are in the path of an incident wave. These methods can be divided into two groups: matrix algorithms, and ray tracing approaches. Each of these algorithms has its individual strengths and weaknesses which are discussed in Chapter 2. Many of these algorithms have been implemented on serial machines, and the design of a parallel implementation can initially use the serial program's structure.

Different methods exist for converting serial programs to a parallel architecture, and these methods also apply to new code generation. As is the case for converting existing code, when generating new programs, care must be taken to ensure that an appropriate method is chosen. Often the "optimal" solution is a combination of two or more methods. Some of these methods involve the way the input data is handled, and others are concerned with the division of labor. These methods include domain decomposition, control decomposition, static load balancing, and dynamic load balancing. These techniques are discussed in Chapter 2.

1.2 Problem Statement

In order to achieve the goal of an "accurate" simulated RCS, an efficient method for predicting the radar scattering from an object is required. For any but the simplest of problems, current implementations take several hours to compute the RCS of a typical scene (18). This particular research evaluates existing programs in computational electromagnetics with a goal of determining the "best" one for a high frequency (8 GHz $< f <$ 20 GHz) application. Factors to consider in the choice of an RCS algorithm are the feasibility of optimizing an existing program to improve its performance and the effort required to convert that software to run on a parallel machine. Various parallel architectures

are examined in order to find a machine that matches well with the proposed algorithm, so that an efficient implementation is possible. Methods for converting existing parallel code to parallel applications are analyzed, and an "optimal" match between the chosen program, architecture, and conversion method is determined. Finally, the resulting application is evaluated for accuracy and efficiency, and the results are compared with other parallelization efforts of different algorithms.

1.3 Summary of Current Knowledge

In order to calculate a simulated RCS, the optical image synthesis model is modified to allow a range of viewpoints and incorporate the effects of diffraction. Optical image synthesis sets pixels (picture elements) of a simulated viewscreen to an intensity and color value. This represents a simulated view of a scene from a single viewpoint. Since an RCS is normally concerned with a few discrete frequencies (lying far below the visible spectrum), color has no meaning. An RCS contains information about the relative intensity (power) of the EM energy radiated in all directions away from the scene as a function of the relative angle. An RCS is typically measured in terms of area (square feet) and is two-dimensional in nature. In order to meet the need of computational solutions for determing the RCS of a scene, the scientific community has created several programs which calculate a simulated RCS, and they are divided into two groups: matrix models and ray tracing models.

Matrix oriented algorithms approach the problem by modeling either the structure itself, or the space surrounding the scene in matrix form. Matrix operations are performed to solve the resulting expressions, and the results are presented in matrix form. These results give the electric and magnetic field strengths in a rectangular coordinate system. The total field strength (or power) can be calculated from these fields and represents the intensity. This data can be converted into polar form to give an angle for later analysis. Some matrix oriented algorithms are Method of Moments (MOM) (7), Finite Difference - Time Domain (FD-TD) (15), and Conjugate Gradient (CG) (2).

Ray tracing algorithms calculate discrete paths from one point in space to another, checking for reflections and other interactions with the objects in the scene. An example of this type of processing assumes a point in space, and a direction of travel called a "ray". The scene geometry is modeled mathematically, and the theoretical ray path (defined by the initial point and direction) may or may not intersect one or more of the objects in the scene. If an object in the scene intersects the calculated ray path, that ray ends, and a new ray is generated based on the characteristics of the interfering object. This new ray has a

direction and starting point different from that of the original ray, but is directly related to that ray through the interaction with the object. Processing of related rays ends when the last ray of the series exits the scene geometry completely. Its information (intensity, and direction) are then stored for later use. Some ray tracing programs in computational electromagnetics are SRIM (11), JET (1), and the Numerical Electromagnetic Code - Basic Scattering Code (NEC-BSC) (13).

Computer programs in general can be divided into categories according to their structure. This dividing value is called the grain of the algorithm (9). Algorithms with a large amount of interaction among data items are said to have a fine grain, while algorithms with little or no dependencies between the individual data items are said to have a coarse grain (10). Other characteristics of programs are their complexity (order-of) and size (6).

In the area of parallel computers, several architectures are in widespread use. Shared memory architectures have many processors which all access the same set of memory, each taking its turn if conflicts arise. In distributed memory machines, each processor has its own dedicated set of memory which it alone can access. SIMD (single instruction, multiple data) machines have a single control element which sends instructions to all the processors, all of which execute the same instruction at any given time (10). The Connection Machine, designed by Thinking Machines Inc., is an example of a SIMD architecture, while the Cray Y-MP, from Cray Research, is an example of a shared memory structure. Hypercubes such as the Ncube/2 (from NCUBE) and the Intel iPSC/860 are examples of distributed memory architecture. The Connection Machine has a very large number of relatively slow processors, and works well with a fine grain application. The Cray Y-MP also works well with fine-grain applications, but does so with a small number of super fast processors The Ncube/2 can have up to 8192 processors, while the iPSC/860 may have up to 128 processors and both generally work best with coarse-grain applications. Each of these machines has approximately the same processing power (within an order of magnitude of each other) and can be thought of as general purpose machines. A new arrival on the parallel computer scene is the Intel Paragon XP/S, and its prototype, the Delta machine, currently at the Jet Propulsion Laboratory in California. The Paragon XP can have up to 2,000 processors and has a processing speed of up to 300 GFLOPS (3).

1.4 Assumptions

It is assumed that the target architecture has a compiler which can efficiently convert the chosen program from source code into object and machine code. In fact, this

requirement is a major factor in the selection of an algorithm and target machine. It is also necessary that sufficient time be available on the target machine for the conversion process including initial analysis, design, code modifications, testing, debugging, and the gathering of performance data once the conversion is complete. It is hoped that access can be gained to the largest possible model of the target machine in order to gather as much data as possible.

1.5 Scope

The investigations of this thesis effort concentrate on the optimization and parallelization of an algorithm or program that can calculate the RCS of a scene. Previous work in the area of parallelization of computational electromagnetics is reviewed. If a particular serial program or algorithm has the capability to perform more than a simulated RCS, those additional capabilities will be left as is, with no conversion performed. The actual method of calculating the results will also be left as is, with no improvement in the accuracy of the results attempted.

1.6 Standards

During any modification of an existing program, and especially during optimization, it is very important that all aspects of the original functionality of that program be preserved unless the researcher wishes to improve on the existing functionality. Side effects of optimization that degrade accuracy or usefulness should be avoided at all costs. At all points, modifications to the chosen algorithm or program should be benchmarked against the original code to ensure that the accuracy and functionality of the program are retained. Any improvements in either area should be carefully documented, and discrepancies noted. During the optimization and conversion efforts, current software engineering techniques should be employed to help ensure that all original functionality is preserved and improved with documentation provided.

1.7 Summary

This chapter has covered the need for research in the area of reducing the time required to calculate a simulated Radar Cross Section of a group of objects. The next chapter covers previous work that is relevant to the subject and scope of this thesis research.

II. Previous Research and Background

2.1 Introduction

The field of image synthesis has developed a great deal since its inception in the late 60's (8). Initially, research in image synthesis was confined to the visual spectrum, rendering views of a group of objects called scenes for visualization that could be displayed on a monitor. Recently, a new branch of image synthesis has evolved: electromagnetic image synthesis, or the calculation of radar cross-sections (RCS). Many different methods for calculating the RCS of an object have been developed, several of which are mentioned in Chapter 1. These techniques are quite varied in their approach and a review of these methods is required in order to provide the necessary guidance for this investigation.

Similarly, a review of available parallel architectures is needed in order to choose the "best" machine for this work. Factors to be considered are the grain of the machine, its applicability to the chosen algorithm or program, and availabililty of access to that machine. Techniques for parallelizing programs and algorithms are covered so that the best possible choice can be made for the conversion to a parallel implementation on the target machine.

2.2 Algorithms and Programs for Calculating an RCS

Two general approaches have been developed for calculating an RCS: matrix oriented, and ray tracing algorithms. Three matrix oriented algorithms that are in current use are Finite Difference Time Domain (FDTD), Method of Moments (MOM), and Conjugate Gradient CG). Examples of ray tracing programs are SRIM, JET, and NEC-BSC. The following paragraphs discuss these techniques and their characteristics.

2.2.1 Finite Difference - Time Domain The FDTD algorithm seeks a solution to Maxwell's equations by tracking the evolution of scattered fields in time. As described by Patterson *et al.* (14),

> *An incident electromagnetic wave propagates into a volume of space gridded as a 3-dimensional lattice containing a conducting or dialectric structure. The wave's interactions with the scattering object are then observed. First the electric field quantities are calculated. Next, using the newly obtained electric field quantities, the magnetic field quantities are updated. Then, using the newly calculated magnetic fields, the electric field values are updated.*

The process iterates until the difference between successive intensity values is less than a specified amount (a steady state is reached). The 3-dimensional lattice is divided into individual cells. The values for the electric and magnetic fields are stored in matrices which can then be manipulated to produce the intermediate answers and eventually, the final answer. Perlik and Opsahl (15) describe FDTD as one of the most robust electromagnetic scattering codes available today. What they mean is that the results are very accurate and applicable to a wide variety of situations. Appendix C contains a more thorough explanation of this method. A more complete discussion of FD-TD may be found in Appendix C, taken from a MS thesis by J. Raley Marek (12).

2.2.2 Method of Moments Method of Moments (MOM) is another name for the Numerical Electromagnetic Code (NEC 2), which should not be confused with the Numerical Electromagnetic Code - Basic Scattering Code (NEC-BSC). The NEC-2 approach was developed at Lawrence Livermore National Laboratory and uses an integral representation for the electric field of a volume (V) current distribution to model thin structures using wire segments:

$$\vec{E}(\vec{r}) = \frac{-j\eta}{4\pi k} \int_V \vec{J}(\vec{r}) \cdot G(\vec{r}, \vec{r}')\, dV \tag{1}$$

The magnetic field is represented by an integral of the surface current distribution J_S.

$$\vec{H}^S(\vec{r}) = \frac{1}{4\pi} \int_S \vec{J}_S(\vec{r}') \times \nabla' g(\vec{r}, \vec{r})'\, dA' \tag{2}$$

An object of interest can be represented by using a number of surface "patches" which, when joined together, would form the desired shape. Figure 3 shows such a model using wire segments. Each patch has its own equation for describing the electric and magnetic fields that are associated with it. When all the equations have been defined, their coefficients can be placed in a matrix, and all the equations can be simultaneously solved using standard matrix manipulations. The system of linear equations can be written as

$$[A][F] = [E] \tag{3}$$

where

[A] is a dense matrix called the "interaction matrix",

[F] is a vector of unknown basis function amplitudes, and

[E] is a vector of the excitation at the center of the wires and/or patches.

From the solution of the amplitudes of the basis functions, one can determine the near- and far-field quantities (14). A more detailed description of these equations may be found in (7).

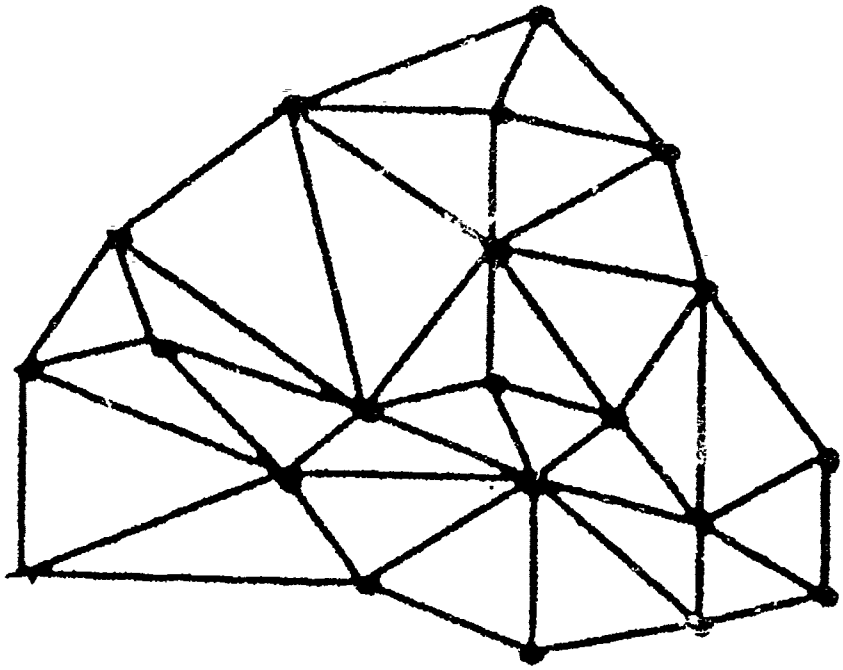

Figure 3. Typical wire-frame model of an object

2.2.3 Conjugate Gradient The Conjugate Gradient (CG) algorithm achieves a non-linear solution time by using the matrix manipulations of inversion and multiplication to arrive at an answer in a finite number of steps, always less than the number of unknowns. For electromagnetics, the resulting equations usually involve convolutions over the unknown current density. The Fast Fourier Transform (FFT) can efficiently evaluate convolution integrals without requiring cumbersome integrations. Including FFT's with the conjugate gradient approach achieves greater accuracy and speed (2). In the CG-FFT algorithm, the object is divided into sections, and equations for the electric and magnetic fields are derived. The number of unknowns in a particular case is chosen according to a

criterion that includes both spatial and frequency domains as well as the requirements of linearity in the convolutions (2).

The general form for this equation is

$$A\left[\vec{J}\right] = \vec{E}^{i} \tag{4}$$

while a general form of such an integral is

$$\vec{E}^{i}(\vec{r}) = \overline{\overline{\eta}}(\vec{r}) \cdot \vec{J}(\vec{r}) + \int_{v'} \vec{J}(\vec{r}') \cdot \overline{\Gamma}(|\vec{r} - \vec{r}'|)\, dv' \tag{5}$$

The coefficients for these equations are then placed into matrices, and the matrices are then manipulated through FFT operations as well as the standard matrix manipulations. These equations are described further in (2).

2.2.4 Evaluation of Matrix Methods A major computational problem of these matrix methods lies in the fact that they all rely on matrices and matrix manipulations to compute the final answer. The dimension of the matrices used is proportional to the frequency of the incident EM energy and the overall size of the object(s) in the scene. As the frequency of the EM energy increases, so does the dimension of the matrices. Likewise, as the volume of the scene increases, the dimension of these matrices also increases further (2). The overall size of a matrix is proportional to the square of its dimension, and normal matrix operations (such as inversion or a matrix multiply) perform a number of operations proportional to the cube of the matrix' dimension (17). For example, assuming that the dimension of the matrices is linearly proportional to the frequency, and holding the size of the object constant, if the frequency of the electromagnetic energy increases by a factor of four, this means that the required matrices must increase by a factor of sixteen and the number of operations increases by a factor of 64. Therefore, lower frequency applications can expect an execution time that is faster than that of a high frequency application. To illustrate, the critical section of an application at 8 GHz would require 64 times as many calculations as that same critical section with an application at 2 GHZ. The critical section is defined as that area of the code that actually does the numerical calculations. This section does not comprise the entire program, but uses up most of the total execution time. Other areas of the program, such as I/O, require a fairly constant amount of time.

2.2.5 Ray Tracing Algorithms The solution time for ray tracing algorithms is not dependent on the frequency of the electromagnetic energy being simulated. Ray tracing algorithms calculate a result based on hypothetical paths that may be drawn (or traced) from a source of energy to a receiver. Ray tracing had its origins in the late 1960s, and calculates the paths for a number of rays. Traditional ray-tracing algorithms calculate the paths for a large number of rays, sometimes exceeding one million in number. For this reason, ray tracing was not seriously developed until faster, more powerful computers were developed to handle the enormous amount of calculations involved (8). Ray tracing algorithms may be divided into two different classes: observer-viewplane ray tracing, and source-destination ray tracing.

2.2.6 Traditional Ray Tracing This thesis refers to the traditional ray-tracing approach as "observer-viewplane" ray-tracing. Observer-viewplane ray tracing has four basic components: a source, an observer (or receiver), a viewplane, and a set of objects called a scene. The source, the observer, the viewplane, and all objects in the scene have a distinct position which is defined in a reference coordinate system. Rectangular, cylindrical, and spherical coordinate systems may be used for various applications. Figure 4 illustrates how this is done.

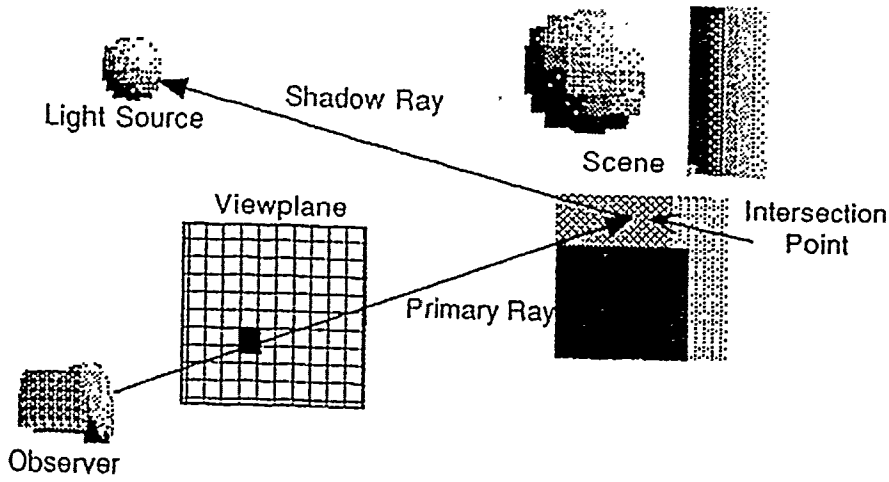

Figure 4. Traditional ray tracing approach

The viewplane is divided into a number of "pixels", and the total number of pixels corresponds to the number of rays which are to be generated. For typical applications, the number of pixels (and thus the number of rays) is very high, usually in excess of 100,000.

The accuracy or size of the image is determined by the number of rays calculated. The higher the number, the greater one or both of these values may be.

To trace a ray, calculation begins with the observer, and rays are traced (drawn) from that point , through an individual pixel of the viewplane, to the scene. Upon arriving at the scene, each ray is processed to see if its path intersects any of the objects of the scene. If an intersection occurs, a new ray is generated with a new direction according to the interaction parameters (angle of incidence and slope of the object at the point of intersection). Each interaction either increases (if the object is a source) or decreases (all other objects) the intensity associated with that ray. If the ray has not encountered a source, eventually the intensity associated with that ray will be low enough to be considered zero. Therefore, after a predetermined number of interactions, processing may cease for that ray since its intensity information is no longer useful. Once a ray has been completely processed, the pixel associated with that ray will have its color and intensity set according to the interactions that the corresponding ray experienced. Carter (8) gives a very good description of this process and shows how such a ray tracing algorithm could be applied in optical image synthesis.

Gustafson, *et al.*(11) used SRIM, a traditional ray-tracing EM image synthesis program, to perform some measurements in parallelization of large programs. In terms of computational complexity, Gustafson's group (11) states that the algorithmic cost (or complexity) of a observer-viewplane approach is proportional to the number of rays fired, the number of reflections allowed, and the number of objects in the scene (since each object in the scene must be checked for further interactions after each intersection). Carter (8) introduces a technique to traditional ray-tracing which reduces the number of objects that are examined for possible intersections, and thus, the complexity of the best observer-viewplane approach is $O(\log n \cdot m \cdot b)$ where n is the number of objects, m is the number of rays, and b is the number of reflections allowed. The number of rays can be as high as one million, with the number of reflections reaching up to 20, and the number of objects can be more than 1000 for this type of ray tracing. Because its foundation lies in optical image synthesis, observer-viewplane ray tracing deals only with reflections from objects without accounting for any other contributions to the result. For optical applications, this produces the effect of very sharp shadows. This effect is not realistic, but in order to make it more realistic, much more effort would have to be expended to somehow model the complex interactions that take place at the edges of the objects in the scene (8).

2.2.7 Diffraction Typical radar frequencies in RCS problems are much lower than the visual spectrum, and the strength of the contributions from edge and corner interactions is inversely proportional to the frequency of the incident energy. This means that for RCS applications, these interactions have a much more pronounced effect on the final result than in optical synthesis. SRIM makes no allowance for corner and edge interactions and also produces a result that is valid for only one direction. These factors make it inadequate for RCS calculations.

The interactions that take place at corners and edges are commonly referred to as "diffraction". Diffraction occurs when a wave of energy strikes the edge or corner of an object. When this happens, the electromagnetic energy doesn't simply stop at the edge of the object. In other words, no clear shadows result. In effect, the edge of the object acts as a weak source, and a new wave of energy radiates out from that edge. The effects of diffraction are directly proportional to the wavelength of the incident energy. Since wavelength is inversely proportional to the frequency, as the frequency increases, the strength of the diffracted energy grows weaker (19). In the visible spectrum (visual light is electromagnetic energy at very high frequencies) the effects of diffraction are so weak that there is only a minimal contribution. This effect, on the other hand, can be quite pronounced at frequencies typically used by radars (2 - 16 GHz).

The only contribution from diffraction in a observer-viewplane approach is when a generated ray strikes an object exactly on its edge. How such a ray is treated determines how well diffraction is handled. In optical ray tracing, that ray either carries on as if no reflection had taken place, or reflects normally. Reflections are handled by generating a new ray with a new starting point and a new direction. The direction of the new ray is determined from the angle the old ray made with the intersecting object. Thus, in optical ray tracing, diffraction has no contribution. In order to accurately account for the effects of diffraction, a observer-viewplane approach would have to treat each edge and corner of each object as a potential source, and generate m more rays whenever a ray struck an object near an edge. This complicates the computations considerably, and the algorithmic cost (or complexity) becomes $O(m^c \cdot b \cdot \log n)$ where c is the number of consecutive diffractions that are allowed. Since the results of diffraction are weaker than reflections, c would be much smaller than b. In traditional observer-viewplane ray-tracing, this approach has never been implemented since the value of m^c is tremendously large. For example, if $c = 2$, and $m = 200{,}000$, the resulting value is 20 billion!

2.2.8 Source-Destination Ray Tracing EM image synthesis algorithms that use the traditional observer-viewplane ray-tracing method are based on the optical versions that preceded them and do not treat the effects of diffraction accurately. A source-destination algorithm addresses the effects of diffraction by changing the method of generating the rays. Instead of calculating ray paths, and then checking to see if a given ray intersects an object, the object-based approach begins with an object and checks to see if a ray can be traced from the source to the receiver (observer) after interacting with that object. This eliminates the need to generate a large number of rays, and each object can be treated for all possible interactions, including diffraction. Multiple interactions are handled by treating each object as a new source, and checking with all other objects for interactions that would result in a path in the desired direction.

Marhefka uses the source-destination ray tracing approach for his implementation of the Numerical Electromagnetic Code - Basic Scattering Code (NEC-BSC). NEC-BSC was developed for applications specific to the proposed space station, and has found use in the United States Navy (13). NEC-BSC achieves some very good results with a serial implementation of the source-destination algorithm, having been validated to -20dB. A weakness of NEC-BSC lies in the the time required for execution. This weakness arises from the computational complexity of the source-destination ray tracing algorithm. Since every possible combination of objects and types of interaction are treated separately, the algorithmic cost is high, $O(n^b)$. If the number of interactions, b, is high enough, the execution time grows tremendously. For this reason, Marhefka implemented interactions with a maximum of three reflections and two diffractions. Also, within this subset, interactions that provided a very weak contribution were left out. In addition to the computational complexity of the source - destination algorithm, preliminary analysis of NEC-BSC has shown that it is largely inefficient in its calculations. Many values are recalculated several times instead of being saved for later use.

2.3 Equipment

The target computer architecture for this work is the Intel iPSC series hypercubes. An iPSC hypercube can have anywhere from 8 to 128 processors (nodes), with the actual number of nodes present in any one machine being equal to a power of two (2^n). In order to accomodate the different configurations possible, the final product is written in such a way that the program may be run on any number of nodes.

2.3.1 Computer Hardware Since their introduction, traditional computers have been following a trend of increasing computational speed and power. Many researchers now believe that this trend will slow down and the computational speed of a traditional single processor will approach a maximum. This maximum is related to the speed of light, and once approached, the speed and power of an individual processor will not be able to increase further (10). As a traditional processor approaches this maximum, its cost grows dramatically, and today's top processors (such as the CRAY) command premium prices on the open market. Parallel computers, which use many processors, can achieve computational speeds much higher than this maximum through combining the speed and power of many processors. Furthermore, since the individual processors need not be top-of-the-line, the overall cost of a parallel computer can be much cheaper than that of the super-fast traditional processor.

An example of the advantage a parallel architecture has over a traditional computer can be drawn form a comparison of the NCUBE/ten and the CRAY Y-MP (11). The NCUBE/ten has 1,024 processors, each of which is based on the architecture of the VAX 11/780. The CRAY has eight processors which represent the state of the art in processing speed. A ray-tracing application was run on both machines with the CRAY executing the program slightly faster than the NCUBE/ten (105 sec vs 124 sec). The same problem when run on a VAX 11/780 took over 35,000 seconds to execute; thus, the speedup available through either machine over the older traditional computer (the VAX) is obvious. Since an NCUBE/ten costs only $1.5 million (compared to $30 million for a CRAY Y-MP), this illustrates the cost-effectiveness of parallel computers. The conclusion is that very similar execution times can be achieved at a considerably lower cost.

The term hypercube means that the computer has 2^n processors, and each processor is connected directly to n other processors. The furthest distance from one processor to another is also n, thus allowing a large number of processors to be connected in an efficient manner. The NCUBE/2 is the next model (after the NCUBE/ten) of parallel computer produced by Ncube, and like its predecessor, uses a hypercube interconnect to link the processors together. The NCUBE/2 can have up to 8,192 processors, and has a theoretical performance of 4 GFLOPS[1].

Another example of a hypercube architecture is the Intel iPSC/860. This parallel computer can have up to 128 processors and has a theoretical maximum performance of

[1]one GFLOP denotes the ability to perform one billion floating point operations per second

7.5 GFLOPS. Each processor, an Intel i860, is a RISC chip operating with a clock speed of 40 MHz and is rated at 60 MFLOPS peak. Recent work has experimentally measured the bandwidth and latency of the iPSC/860 when sending messages between the nodes (21). In addtition, Intel has recently announced a new parallel machine based on the i860 microprocessor (also used by the iPSC/860) called the Paragon XP/S computer. This new machine has a peak power of 150 GFLOPS and uses a mesh interconnect instead of a hypercube. Applications which run on the iPSC/860 are expected to run on the new machine with only minimal conversion work required. Intel has also developed an efficient message-passing network that connects the processors together. This network allows messages to proceed to a distant destination without interrupting any of the other processors that lie between the sender and the receiver (5). Currently, the Target Recognition Branch of the Avionics Directorate at Wright Laboratory, Wright-Patterson AFB, OH has an eight node model of the iPSC/860 which is employed in this research.

2.3.2 Parallelization Techniques Some of these techniques are domain (data) decomposition, control decomposition, static load balancing, and dynamic load balancing.

2.3.2.1 Data Decomposition In many programs, the individual items of data to be processed have, by design, a built in independence, which would allow separate items of data to be processed simultaneously with no side effects which might arise if data dependencies were present. For example, consider a loop in a section of serial code where the results of the data processed in each pass are not dependent on earlier calculations or comparisons from previous passes through the loop. An application with this sort of independence would be a ray tracer. The handling of each ray is completely independent of the results of other rays. Thus, each ray can then be calculated in parallel, or simultaneously with all other rays. A simple data decomposition would divide the iterations among the available processors, assigning N/n rays to each processor where N is the number of rays (typically very large), and n is the number of processors. This kind of implementation is called data decomposition. Since each node operates on separate items of data, every node must be able to completely process each item of data, and thus, a complete set of the code to be executed must be loaded onto each node.

2.3.2.2 Control Decomposition Sometimes, data dependencies do not allow partitioning the data set among the processors since such a decomposition would result in prohibitive communication costs. In such cases it may be more advantageous to divide

up the various tasks among the processors. In other words, assign the task modules in the program to the various processors. This kind of decomposition is similar to a task scheduling problem, and care must be taken to assign the modules in an efficient manner. Communication would be required to pass parameters back and forth, but this cost is normally less than that of a data decomposition of the same problem. This type of implementation is called a control decomposition, and each node has a unique set of code to be executed. One case of control decomposition is task scheduling on multiple processors; each node has a unique set of code (its current task), and executes that program. In more involved cases, the tasks become interrelated modules of the same program, and additional communication is required. This additional communication, however, is normally only a moderate amount.

2.3.2.3 Combinations Sometimes it is useful to use both control and data decompositions for a particular application. Such a scheme would have several groups of processors assigned to unique tasks, with the data divided among the groups as well. Thus, for example, a pipelining effect could be realized: each processor in the first group would begin working on its corresponding first item of data; when done, it would then pass along an intermediate result to a corresponding processor in the next group of processors for further work and then begin working on the next item of data. Each processor in each successive stage would process the intermediate results sent to it, and send its result on. The final group of processors would then store the final result. Such an approach was used by Gustafson *et al* (11) at Sandia National Laboratories with a ray tracing package (SRIM 2.2Q). This approach is also useful when the entire program to be executed is too large for the available memory on a single node. By dividing up the modules, the executable code given to each processor is less. The data are also still divided among the first group allowing the software engineer to take full advantage of the parallelism in a particular application.

2.3.2.4 Load Balancing When converting a serial program to run in a distributed memory environment, a major objective is to have some way to balance the execution time among the processors thus minimizing idle time. Theoretically, each processor should do exactly the same amount of work in order to achieve maximum efficiency and the best possible speedup. In other words, the work load needs to be balanced among the active processors. Static load balancing is a technique where the programmer hard codes the division of labor among the nodes, specifying exactly what each node will do before

the program is executed. Dynamic load balancing occurs when the program itself divides up the work according to heuristics that have been written into the balancing routine.

2.3.2.5 Static Load Balancing When a software engineer uses static load balancing, in order to efficiently partition the work, some prior knowledge of the complexity of the data set is required in order to make a good estimate. A simple scheme for static load balancing (when the data is handled in a loop from 1 to N) is to let each node do $x = N/n$ items of data; each node handling the items from ix to $ix + x$ (i representing the current node number) in the data set where n is the number of nodes. This approach may not yield an efficient implementation, however, if some of the data items require more processing than others. Another approach used in some ray tracing programs, for example, is to have each node iterate through the entire data set with an increment equal to the number of nodes. Thus each node would process data items i, $n + i$, $2n + i$, $3n + i$, ... , $N - n + i$. This approach attempts to compensate for an unequal amount of processing by assuming that neighboring items have a similar amount of complexity associated with their computation. By assigning neighboring items to different processors, those items with more complexity are spread out among the available processors, and similarly the less complex items are also spread out evenly among the nodes.

2.3.2.6 Dynamic Load Balancing Dynamic load balancing occurs as the program is running and can compensate for differing levels of complexity throughout the data set. One type of dynamic load balancing divides the available nodes into two groups by purpose: master and slave nodes, also known as controller and worker nodes. Each worker node receives a relatively small data set to operate on, and when done, it then notifies its controller node that it is ready to operate on another set of data. Here, the generic term "set of data" can be thought of as either individual data items, or possibly even separate tasks that are to be executed. The master (or controller) node then determines the contents of the next set of data that the requesting slave (or worker) node should execute, and forwards that information to the requesting node. This process repeats itself until the global data set has been entirely processed. With a centralized list, one master controller maintains the global data set and distributes the work among the worker nodes. One controller, however, can only manage a finite number of nodes before the communications from its worker nodes begin to cause a bottleneck. When this happens, the controller is unable to respond quickly to a worker node's request, and overall performance suffers as the slave nodes wait for the next set of data. The actual number of worker nodes that a

manager node can efficiently handle is dependent on the grain size of the target machine and the average amount of time spent processing each set of data. If the number of nodes available becomes large enough to become a problem, then an intermediate level of worker nodes can be used under the master node. Alternately, a number of controller nodes can maintain a "distributed list", with each controller node sending out data sets from its portion of the overall list. If a controller node exhausts its local list, it can ask one of its neighboring controllers for more work. Once all the local lists have been completed, the program then terminates.

2.3.2.7 Efficient Dynamic Load Balancing Normally, dynamic load balancing has some overhead associated with implementing the heuristics and message passing. This overhead arises when the requesting node waits for a response from its controller. Here, the sending node is idle, awaiting the next item of work. Once the return message arrives, the sending node continues on, processing the data items in the new set. If the items of work can be subdivided into recognizable data items, some of this idle time can be eliminated or "hidden" in the overall execution time. Such a division would allow the worker node to send a request for more work before it actually finishes its current set. The request would travel to the appropriate controller, and that controller would send the next item of work to the requesting worker node. While all this is occurring, the worker node is finishing its current set of data, and does not wait for the next item of work upon completion. If, for example, a ray tracer program divides the work into groups of four rays per "item", When the worker node has completed three of those rays, it can then send a request for more work. While the worker processes the last ray, the controller receives the message, determines the next group of rays to be processed, and sends a message back to the requesting node. For greatest efficiency, the time required to process one data item (in an item of work) must be greater than the total time between sending the request and the arrival of the next item of work. Otherwise, more data items must be included in a group, and the request must be sent before the penultimate data item. A balance between the round trip message time and the processing time must be reached such that the message time is less than the processing time in order to achieve the best possible efficiency.

2.3.3 Examples of Parallel Implementations Suhr improved the execution time of NEC-BSC by parallelizing the existing serial code using a static load balancing technique with a domain (or data) decomposition. In this work, for some simple test cases, he demonstrated that gains are possible. He measured an improvement over the VAX 11/780

of about 6.1 for one node (18). Unfortunately, he was unable to achieve good time efficiency with a limited number of nodes.

Gustafson and Co. were able to show that a dynamically load-balanced application of SRIM allowed them to achieve near-linear speedup in the execution time. Speedup is the ratio of the execution time (on a traditional computer) for the best serial implementation of an algorithm to the execution time for n processors in a parallel machine. Ideally, this value should equal n showing 100% efficiency (18). Often this is not the case, because overhead expenses associated with the unique features of a parallel computer add to the execution time of the program. This decreases the speedup to something less than n. The work of Suhr (18), and Carter (8), in their work with the Intel iPSC/2, also reflects this phenomenon.

2.4 Conclusion

If accurate calculations of a high frequency RCS are to be accomplished in a cost-effective manner, an efficient, accurate algorithm needs to be developed for a cost-effective computer. Matrix oriented algorithms such as FDTD, MOM, and CG become very time consuming when the frequency of the incident EM energy increases. Additionally, observer-viewplane ray tracers such as SRIM are inadequate because of the single viewpoint and the inability to handle the effects of diffraction. NEC-BSC performs the serial calculation of an RCS with good accuracy and is an excellent candidate for optimization because of its many inefficiencies. Parallelization should result in the greatest gains in execution time and is the primary focus of this thesis effort.

Recently, four researchers using an Intel iPSC/860 hypercube won an award for price/performance results with calculations in superconductor structure (4). This shows that cost-effective computing is possible with parallel computers. This was only achieved after a great deal of work in optimizing an existing program and converting it to run on the parallel machine. The Intel iPSC/860 was chosen as the target machine for this work because of its potential for performance, and availability. Another consideration was the relationship of the iPSC/860 to the new Paragon parallel computer which should be available in the near future. Since the next generation uses the same type of processor, and the differences between the architectures will be largely transparent to the user, any program that executes on the iPSC/860 should execute on the new machine, providing a great deal of power to the user.

Because the iPSC/860 is a coarse grain machine, a data decomposition is chosen for the implementation which should provide the best possible division of labor among the available processors. A control decomposition would probably be too coarse for the iPSC/860, and an uneven work load could easily result. Additionally, a dynamic load balancing technique is implemented since it offers the greatest possibility for even load balancing under all conditions. Exactly how to decompose NEC-BSC onto the target machine is determined in Chapters 3 and 4 as the structure of NEC-BSC is examined.

2.4.1 Software Development Environment, Tools and Techniques The development of the "windowing" environment for workstations has given the software engineer a very effective tool for all stages of an applications development. By logging into the target machine with multiple windows, several activities can be done concurrently. for example, while an update to the baseline is being compiled (sometimes a lengthy process) in one window, results from a previous version can be analyzed. During long execution times (while gathering data) in one window, an analysis of any previous data can be typed into a document. Also, if errors should be detected during execution in one window, a search for the code that caused the error can be performed in another window, keeping the symptoms of the error in view without the need to print out error listings. Consequently, a workstation with the windowing environment is chosen for use in the development of the application. The workstation used was the Sun Sparcstation 2 using Openwindows.

2.4.2 Summary This chapter covers previous work in electromagnetic image synthesis, detailing efforts in matrix and ray-tracing models. Candidate computer architectures for a parallelization of a chosen algorithm are put forth, and techiques for parallelization are explored. The next chapter initiates the implementation process, analyzing a serial implementation of the chosen algorithm, and presents a high-level design.

III. Analysis and High Level Design

3.1 Introduction

The analysis and design of a parallel implementation of NEC-BSC is the objective of this research. Accordingly, the code is first analyzed using software engineering techniques, and its structure is then modeled with UNITY, a high level design language. Inherent parallelism in the structure of NEC-BSC is identified, and a parallel high-level design is presented.

3.2 The Numerical Electromagnetic Code - Basic Scattering Code

NEC-BSC is a large (> 20,000 lines of code) program that calculates the electromagnetic energy radiated outward from a target scene. The radiated energy can be computed for a large number of parameters, and the output can be in several forms, including near and far zone results as well as the electric and magnetic field strengths received by an antenna, or radiated outward in any given direction. The program can iteratively step through any given series of angles with a large selection of orientations available. Angles are based on a spherical coordinate system and coordinates represent volumetric and circular angles. The far zone results of NEC-BSC take the form of a radar cross section. Therefore, this research focuses on the far zone functionality of NEC-BSC, and all parallelization efforts concentrated on converting the far zone code to run on the iPSC/860. Other features of NEC-BSC such as near zone results and antenna effects were not converted and currently do not execute in the parallel implementation.

3.3 Analyzing the Code

In order to efficiently parallelize an existing application, the software engineer must be familiar with the structure of that program. Modules and interconnectivity can significantly influence a parallel design.

3.3.1 Structural analysis of NEC-BSC Design recovery techniques of software engineering allow a software engineer to recover the structure of an existing program. The User's Manual for NEC-BSC provides a high level structure chart showing the main details of NEC-BSC's structure (13). Figure 5 shows this initial high level structure with nested loops depicted by successive indentations. For a far zone single source application

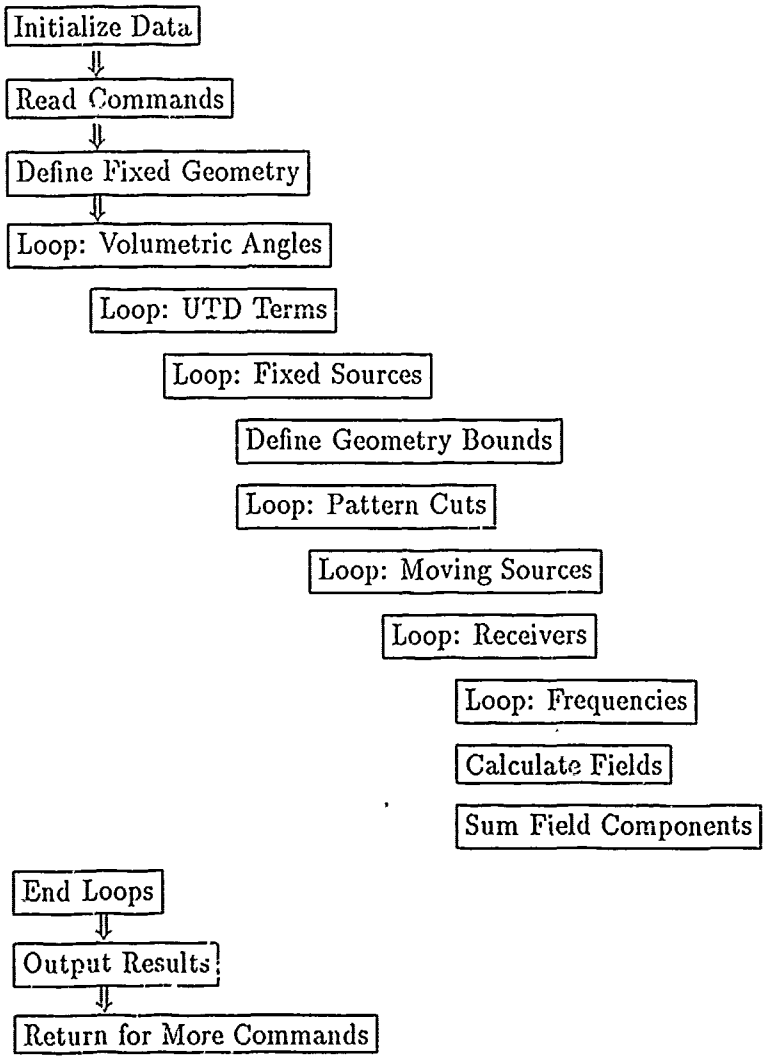

Figure 5. Block Diagram of NECBSC Version 3

with only one frequency, the initial high level structure reduces to that shown in Figure 6. The block depicted in Figures 5 and 6 as Calculate Fields can be further expanded to show more detail. Figure 7 shows this result, depicting the various calculations that are performed.

The structure depicted in Figure 6 shows one inefficiency: calculating the same real angle a multiple of times. This structure recalculates the real angle corresponding to the

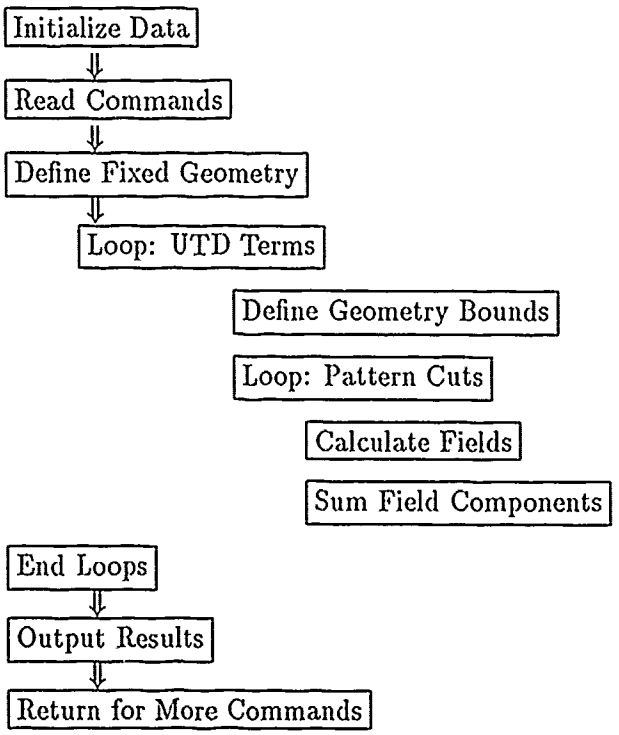

Figure 6. Far Zone Block Diagram of NECBSC Version 3

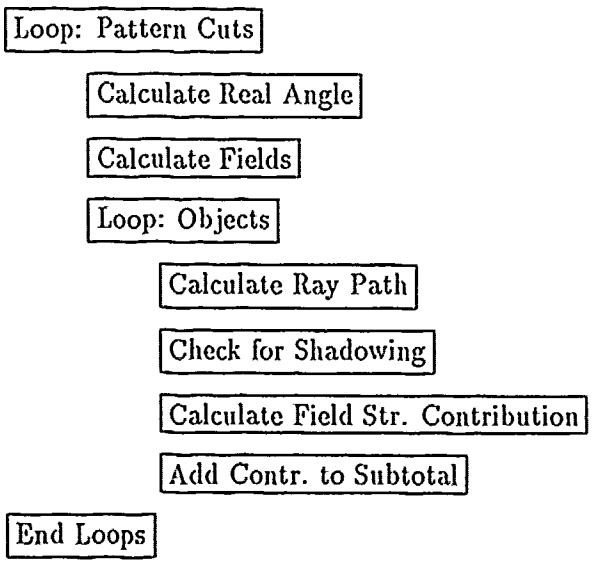

Figure 7. Structure of Calculate Fields

current loop index once for each UTD term[1] (which has a earlier loop construct). Since this calculation is done with a subroutine call, for 1,000 rays, this results in multiple subroutine calls that are completely unnecessary. Moving the pattern cut loop outside the UTD term loop would result in a savings in subroutine calls approximately equal to the number of UTD terms times the number of rays.

Further in-depth analysis shows that each individual UTD term has its own subroutine for the calculation of any appropriate field strength contribution. In order to call the correct subroutine, a very large IF-THEN-ELSE IF structure inside the Calculate Fields block determines what the current UTD term is and calls the appropriate subroutine. Due to the limitations of time, this structure can not be consolidated or optimized for better performance.

Structural analysis reveals another factor which would limit an attempt to improve the efficiency of NEC-BSC. This factor involves the coupling of the program's modules. Most of the modules in NEC-BSC are very tightly coupled and have a loose coherency. The tight coupling arises from the fact that the subroutines of NEC-BSC have a large number of parameters and also access numerous global variables stored in COMMON blocks. Such tight coupling greatly hinders any restructuring or improvements, because of the danger of side-effects (a statement executed locally affecting values that are used elsewhere in the program).

Using the high-level structure provided by the User's Manual and analyzing the code itself, a good picture of NEC-BSC's top level structure results. In order to use this information, this structure must be modeled in a way that shows its characteristics clearly. Chandy and Misra propose a high level design language that can be used to model both parallel and serial programs (9). This high level language is called UNITY, and once an algorithm has been correctly modeled, many details of its structure can be illustrated. Here, the high level organization of NEC-BSC is set out and from the high level design, a parallel implementation can easily be derived. The advantage of describing a serial program in UNITY is that any parallelism that already exists within the structure of a serial program can be easily identified once the UNITY design has been completed. After analyzing the UNITY program, and the inherent parallelism of the application has been identified, a detailed design can be constructed that builds on these parallelisms, and

[1] a UTD term refers to a unique sequence of objects and interaction types. Reflection and diffraction are the two possible interactions, and each may occur any number of times in any sequence in a UTD term. NEC-BSC uses only 20 terms in its loop.

maximizes the gains that can be achieved through parallelism. Additional features map the high level design to the individual machine for which the application is intended. This mapping is called a low-level design and is discussed in Chapter 4.

3.3.2 Serial Design The far zone portion of NEC-BSC is now modeled in UNITY. This design incorporates the far zone functionality of NEC-BSC, assuming one stationary source and no antennas. This removes much extraneous detail pertaining to the other functions of NEC-BSC that would clutter the design here.

Program Serial NEC-BSC

Declare

```
phi, theta : Integer {for use as loop indices}
UTD_num, obj_num : Integer {for use as loop indices}

    {A function for calculating partial field strength in a given direction}

Function Partial_Strength (phi, theta, obj_num, UTD_num) return Complex

i, j, k, l : Integer {Iteration index variables }

pair : array (1 .. 3) of Real {for 3D coordinates }

object : Record{contains the essential information about one plate}
    corners : Integer
    corner_pts : array (1 .. corners) of pair
end record

num_objects, num_terms : Integer

    {Start and stop points for the volumetric and circular loops}

begin_vol_point : integer
end_vol_point : integer
begin_circ_point : integer
end_circ_point : integer
```

{Procedure to read the data in form disk and initialize key variables}

Procedure Cread (num_objects, vol_points, circ_points)

{Variable which will hold the final results}

field_strength : array (1 .. vol_points, 1 .. circ_points) of complex

Initially

{Initialize the field strength and the number of UTD terms}

num_terms = 20

⟨∀ i: begin_vol_point < i ≤ end_vol_point ::
⟨∀ j : begin_circ_point < j ≤ end_circ_point ::
field_strength (i, j) = 0.0⟩⟩

Assign

{Calculate each partial contribution and sum with other calculations}

⟨∀ i : begin_vol_point < i ≤ end_vol_point ::
⟨∀ j : begin_circ_point < j ≤ end_circ_point ::
⟨∀ k : 0 < k ≤ circ_points ::
⟨∀ l : 0 < l ≤ num_objects ::
field_strength (i, k) := field_strength (i, k) +
Partial_Strength (i, k, j, l)⟩⟩⟩⟩⟩

End {Serial NEC-BSC}

In the serial design of NEC-BSC, the variable "UTD_num" holds an integer value that corresponds to the current UTD term. A UTD term is a unique interaction sequence that objects in the scene apply to an incident ray. Examples of UTD terms are single reflection, single diffraction, double reflection, diffraction-reflection, triple reflection, etc. The number of possible UTD terms is represented by the variable "num_terms". The variable "obj_num" holds an integer corresponding to the number of the current object under consideration. The function "Partial_Strength" takes as parameters the current final angles (in a spherical coordinate system), the current object number, and the current UTD term. This function is part of NEC-BSC, and its inner functions are not modeled at this level. Given the input parameters, the Partial_Strength function calculates the

contribution of the current UTD term to the total field strength in the current direction. Since there are 16 different UTD terms, the contribution of each term must be summed to arrive at the total field strength in a given direction. The procedure "Cread" reads in the input parameters from a file stored on disk. The field strength is calculated using complex values (a real part and an imaginary part) and stored in the array "field_strength".

3.3.3 Parallel Design As can be seen from the UNITY design of the serial version of NEC-BSC, a high degree of parallelism exists in the program structure. Several possibilities exist for division of labor among multiple processors. Each of the iterations could be partitioned among the processors of a parallel machine. Some combination of loop values could also be distributed. Using this serial design, a parallel high level design can be drafted to illustrate some of the design decisions at this level. The parallel version of the high level design shows the inherent parallelisms within NEC-BSC even more clearly. In this listing, only the differences from the serial version are shown. The parallel version essentially adds the control and decision features which allow statements to execute in parallel.

Program Parallel NEC-BSC

Declare

{ same as serial version with the following addition: }
{Variable for controlling the calculation of the field strength}
element_done : array (1 .. vol_points, 1 .. circ_points, 1 .. 20, 1 .. num_objects) of Boolean

Initially

{ same as serial version with the following addition: }
{initialize the controlling variable}

⟨∀ i : 0 < i ≤ vol_points ::
⟨∀ j : 0 < j ≤ circ_points ::
⟨∀ k : 0 < k ≤ num_terms ::
⟨∀ l : 0 < l ≤ num_objects ::
element_done (i, j, k, l) = false⟩⟩⟩⟩

Assign

{set each corresponding element in the controlling variable as each}
{contribution to the partial strenght is calculated}

```
[](∀ i : 0 < i ≤ vol_points ::
   [](∀ j : 0 < j ≤ num_terms ::
      [](∀ k : 0 < k ≤ circ_points ::
         [](∀ l : 0 < l ≤ num_objects ::
            field_strength (i, k) := field_strength (i, k) + Partial_Strength (i, k, j, l) ||
            element_done (i, k, j, l) := true
               if not element_done (i, k, j, l)))))
```

End {Parallel NEC-BSC}

Fixed Point

```
FP ≡ (∀ i : 0 < i ≤ vol_points ::
        (∀ j : 0 < j ≤ circ_points ::
           (∀ k : 0 < k ≤ num_terms ::
              (∀ l : 0 < l ≤ num_objects ::
                 element_done (i, j, k, l) = true))))
```

One aspect of UNITY is that all statements in an "Assign" block execute in parallel and each statement executes infinitely often. Here, however, only one calculation per unique combination of parameters is wanted. Accordingly, the parallel version of the UNITY design introduces a new variable called *element_done*. Initially, all the elements of this array are set to "false", and the partial strength contribution of a unique set of parameters is only be added to the sum if its corresponding element in *element_done* is false. Once the calculation has been accomplished, the corresponding element in *element_done* is set to true. This guarantees that each unique contribution is calculated exactly once, even though the statement executes infinitely often.

In the parallel design, all the variables of the serial version are used. Therefore, only the additional variable is shown in the parallel design listing. Similarities in the initialization of these variables are also left out of the parallel version.

An important part of a UNITY parallel design is to show that the program terminates in an orderly fashion by including a final condition called a fixed point (FP) (9). If the FP evaluates to be true, then the program is guaranteed to terminate. This constitutes a proof of correctness for the parallel features of the UNITY design. This proof does not guarantee

the functionality of the parallel program, merely that it terminates in an orderly fashion. The functionality of the program is dependent on the equations used when performing the calculations, not the parallelization of the program.

3.3.4 Proof of correctness In UNITY, each statement executes infinitely often, so in order to prevent the same ray from being calculated multiple times, a new variable was introduced into the parallel version to indicate whether a particular ray had been processed or not. This variable, *element_done*, is a four-dimensional array with each dimension corresponding to one of the nested iterations in the parallel program. Initially, each of the elements of the array is set to false, and when a ray has been processed, its corresponding element in *element_done* will be set to true. In order to guard against multiple calculations of the same ray due to the same statement being executed more than once, a guard is placed on the processing statement. If, at any earlier point, that ray had already been processed, the guard prevents a second calculation. Since each combination of values for the nested iterations is guaranteed to be processed at least once, each element in the variable *element_done* is set to true. This guarantees that the fixed point stopping condition is met, and the program terminates normally.

3.4 Conclusions

Each of the levels of the nested iteration have the potential for parallelization. The volumetric and circular loops have the potential for covering a wide range of values, up to 1801 discrete angles from 0 to 360 degrees. These values could be partitioned widely in a fine grain machine. The UTD terms and objects are few in number and could be distributed among the processors of a coarse grain architecture. The detailed decision of how and what to parallelize is discussed in Chapter 4. As can be seen, there are at least four parameters (volumetric points, circular points, UTD terms, and objects) which could be partitioned among available processors if a data decomposition were to be used. A control decomposition, on the other hand, would require a different sort of analysis than is done here. Such a decomposition would need to analyze the interconnectivity of the separate modules of the program to determine the best way to partition them among the available processors.

3.5 Summary

This chapter presents an initial analysis of NEC-BSC and details the high level structure of the program using a high-level design language called UNITY. Possibilities for parallelization are identified, and a high-level parallel design is shown. The next chapter builds on this work, introducing a detailed, or low-level design, also using UNITY. This low level design has more detail about the eventual implemetation, and implements decisions made during the design process.

IV. Detailed Design

4.1 Introduction

The process of converting an existing high-level design into executable code can be accomplished in a variety of ways. A technique is selected and the high-level design presented in Chapter 3 is refined and expanded into a detailed design, incorporating features of the target architecture in the process.

4.2 Decomposition Technique

Based on the analysis in Chapter 2, a data decomposition technique was chosen to be implemented in the parallelization effort. In order to create a detailed design, however, more analysis of the existing program is required. This is necessary because the object chosen for decomposition should fit the proposed uses for the parallel implementation. This investigation focuses on suitability of the various possibilities for decomposition identified in the high level design. These possibilities are the volumetric angles, the circular angles, the UTD terms, and the objects in the scene geometry. An analysis of the most commonly used features of NEC-BSC reveals that most of the time, the volumetric angle data is not varied, reducing that iteration to one pass. Therefore, a partitioning of that level of the nested iterations is not possible. Further analysis shows that the calculation of the partial contribution due to the separate UTD terms varies greatly in their execution time for the same angles. The UTD terms are also relatively small in number (only six for flat plates up to a maximum of twenty), giving no possibility for scalability to large numbers of processors. The objects themselves are a possibility, but an investigation of that avenue indicates that a great deal of interaction occurs between the different objects, so a full decomposition results in a great deal of communication. For a coarse grain architecture, this means an unacceptably low efficiency. The circular angles (the final possibility), are well suited to data decomposition. Most of the time, the input data requests a full 360 degree scan for the circular angles with varying amounts of precision within the scan. The actual number of discrete angles usually varied between 360 and 1800. With 1800 discrete angles, the actual difference between successive angles could be as small as 0.2 degrees.

4.3 Decomposition Object

Another important factor in determining how to approach a detailed design is the target machine for which the application is intended. In this case, the Intel iPSC/860 is

chosen as the target machine, and its characteristics then influenced the choice for decomposition. In fact, this is why the objects in the scene are not chosen for distribution among the processors. Previous work used the iPSC/2 to implement a static load balanced version of NEC-BSC (18). The Intel iPSC/2 and iPSC/860 are coarse-grain parallel processors with the number of processors equal to a power of two. With a latency of 75 μseconds per message, and a transmission rate of 2.88 Mbytes per second (compared to a peak rate of 40 MFLOPS) (21), an application on the iPSC/860 should avoid large numbers of messages. For this reason, a full decomposition of the objects would not be efficient. This leaves either a partial decomposition of the objects or the circular angles. The number of objects that NEC-BSC can handle is currently limited to 36 (13). This places a limit on the number of processors over which the problem can be partitioned. The circular angles are then left as the best candidate for decomposition across the processors since results for individual angles may be computed independently of other angles. Thus, the processors do not need to communicate among one another while the program is running. The only communications required are the gathering of output data, and the distribution of the work.

4.4 Load Balancing

Once the method of decomposition and the term to be decomposed have been chosen, another factor needs to be considered: load balancing. Even though a fairly equal number of angles can be distributed to the available processors, there is a possibility that some groups of angles take longer than others when calculating the radiated electromagnetic power. It is not desirable for the processors to have execution times that are imbalanced, since the entire application doesn't terminate until the last processor has finished. In order to reduce the execution time as much as possible, the individual processors should all finish at approximately the same time. If one processor finishes significantly later than the other processors, those other processors are idle while they wait for the last one to finish. Idle time results in reduced efficiency, since more time is being taken to produce the same results. Earlier work in parallelizing NECBSC used a static load balancing scheme for distributing the work load (18), and this work concentrates on a dynamic load balancing algorithm.

Dynamic load balancing techniques separate the available nodes into two types: worker (or slave) nodes and controller (or master) nodes. Each worker node is given an initial set of data, and when it finishes this set of data, it notifies its controller that

it is ready for another set of data. The controller maintains a list of all the work that is to be accomplished and sends out new data sets to the workers as requested until all the work has been exhausted. The controller then tells the worker nodes to shut down (20). Dynamic load balancing works well when the work to be accomplished has a wide variation in complexity (9). If succeeding data items do not have similar execution times, then a statically load balanced application would suffer in some cases from an imbalance in the work load. Dynamic load balancing seeks to alleviate this situation by introducing a controller to monitor the status of the job and allocate more work to nodes that would otherwise finish sooner than others.

The design of this basic dynamic load balancing algorithm is fairly simple, but in order to optimize the performance and reduce the overhead associated with dynamic load balancing, some features of the target system must be taken into account. UNITY, a high level design language provides an excellent way to specify such a design so that it can be used in many different parallel environments. Section 2.3.2.6 explains the execution of a dynamic load balancing algorithm and lists alternatives for implementation.

4.5 UNITY Design

4.5.1 Controller

Program Dynamic_Balancer

Declare

Procedure Parallel_NECBSC$_i$ (begin_circ_point, end_circ_point)

{Variables for parallel implementation}
processor : Integer
curr_circ_point : Integer
data : File

{variable for holding the values for the next data set to be sent out}

temp_begin_point : Integer
temp_end_point : Integer
step_size : Integer

num_processors : Integer

Initially

Cread (begin_circ_point, end_circ_point)
curr_circ_point = begin_circ_point

Always

temp_begin_point = curr_circ_point
temp_end_point = temp_begin_point + step_size

Assign

{Assign the next data set to the appropriate processor}
[]⟨∀ curr_circ_point : 0 ≤ curr_circ_point ≤ end_circ_point - step_size ::
⟨[]begin_circ_point$_{processor}$:= temp_begin_point||
end_circ_point$_{processor}$:= temp_end_point||
curr_circ_point := temp_end_point + 1⟩⟩

[]⟨[] begin_circ_point$_{processor}$:= temp_begin_point||
end_circ_point$_{processor}$:= end_circ_point||
curr_circ_point := temp_end_point + 1

if temp_end_point > end_circ_point ∧ curr_circ_point ≤ end_circ_point⟩

⟨∀ processor : 0 < processor ≤ num_processors ::
[]begin_circ_point$_{processor}$:= 0||
[]end_circ_point$_{processor}$:= 0⟩

End {Dynamic Load Balancer}

Fixed Point

FP ≡ curr_circ_point > end_circ_point

Here, the subscripts denote individual instantiations of the indicated procedure on each and every node. The procedure Parallel_NECBSC is therefore the application program as adapted to the target hardware. Temporary start and stop points are set up for use by the controller. When the program is invoked, the local variables begin_circ_point and end_circ_point are set to the current temporary values. These temporary values are simultaneously set to the next values in the data set. When the data set is exhausted, the local values are set to zero.

The load balancing routine mounts above the main program, and essentially calls the main program on each node, giving it a new start and stop point each time it is invoked. In other words, it acts as a supervisor, controlling the execution of the program itself.

4.5.2 Main Program Changes Another necessary change when implementing a dynamic load balancing technique is to modify the main program to cause it to send a message to the controller when it has finished its current set of data. The parallel program now looks like this (only the changes have been listed):

Program Parallel_NECBSC$_i$

Declare

{Declares the dynamic balancing program and a necessary function}
Procedure Dynamic_Load_Balancer (processor)
Function Nodenum() Return Integer
{variables required for local processing and message passing}
my_number : Integer
temp_FP : Boolean

Initially

temp_FP = false

Always

my_number = nodenum()
{Declare the point at which a message should be sent}
temp_FP = ($\forall$ i : 0 < i $\leq$ vol_points ::
$\forall$ j : 0 < j $\leq$ circ_points ::
$\forall$ k : 0 < k $\leq$ 20 ::
$\forall$ l : 0 < l $\leq$ num_objects ::
element_done (i, j, k, l) = true)

Assign

{Send a message if the criterion for signalling has been met}
processor = my_number if temp_FP

Fixed Point

FP $\equiv$ temp_FP $\bigwedge$ begin_circ_point = 0

The main algorithm (for the worker nodes) signals the dynamic load balancer when it finishes its current set of data. When Temp_FP evaluates to true, this message is sent by setting the variable "processor" to the node number that is sending the request. After invoking Dynamic_Balancer, it waits until signaled by the controller to begin processing a new set of data.

The actual algorithm implemented signals the load balancing routine when all but one of the rays in the data set has been processed. This allows the calculations to continue while the message travels to the controller. The details and benefits of this modified dynamic balancing technique are discussed in section 2.3.2.7.

Appendix A gives some examples of actual code that is developed using this dynamic load balancing techniques. The essentials of both the controller and the node programs are shown.

4.6 Summary

This chapter discusses the detailed design decisions that affect the shape of the implemented program. The reasons for choosing a data decomposed, dynamically load balanced approach are given, and UNITY is used to show how those decisions and the charactersistics of the target machine molded the high-level design into an implementable low-level design. The next chapter discusses the results of this design and the conversion of design into implementation, giving timing results for serial, and parallel versions of NEC-BSC.

V. Performance Analysis and Results

5.1 Introduction

The results of the performance evaluations on both serial and parallel versions of NEC-BSC are covered. The serial code is executed and the timing figures are presented for analysis. Timing values for a previous parallel version of NEC-BSC are also shown. These values are the result of executing the previous parallel version with new, more complex data sets. Finally, the execution times for the dynamically balanced version of NEC-BSC are presented and compared with both the serial and previous parallel versions.

5.2 Performance Analysis of Serial NEC-BSC

In order to establish a reference from which the parallelization results can be judged, evaluations of the execution time for the serial version of NEC-BSC are required. Such measurements can also give an idea as to which area of a program uses the most CPU time. Such data is useful to a software engineer who wishes to improve the efficiency of a serial program through optimization of existing code. In such cases, the target machine is usually another serial machine, often the same one initially used to execute the code. NEC-BSC, fortunately, has been validated by independant researchers as to computational accuracy.

One tool which can be used to analyze the performance of existing code is called FORGE and is provided with the iPSC/2 and iPSC/860 hypercubes. FORGE also has a limited automatic parallelization ability. Suhr tried to use FORGE to provide some data on NEC-BSC's performance and gain insight into the process of parallelization of serial code. Unfortunately, FORGE did not recognize variables that used the COMPLEX data type. These variables are used by NEC-BSC to calculate and store the value of the field strengths, and FORGE's inability to handle these variables rendered it useless (18). NEC-BSC was then ported to a VAX minicomputer, and the Performance and Coverage Analyzer (PCA), provided as a part of the VAX toolset, was used to gather data on the performance of the original serial version. Parts of a sample output file created by NEC-BSC is included in Appendix B for reference. The results shown in Table 1 illustrate that for simple example problems, one subroutine, PLAINT, used up most of the time. Since a normal application would involve much more complex scene compositions than the first example, more example data sets were created, each more complex than the previous. These more

complex data sets increased the number of plates by multiples of eight. Table 2 shows that the percentage of time devoted to the PLAINT subroutine increased dramatically as the complexity of the target geometry increased. For the most complex data set, PLAINT was called over one million times!

Table 1. Division of effort for a sample data file

Subroutine Name	Data Count	Percent
PLAINT	6,528	38.2
RPLDPL	2,642	15.5
DPFTWD	1,531	9.0
PDLRPL	1,302	7.6
FLRD	947	5.5
DIFPLT	777	4.6
FLDR	738	4.3
SOURCP	440	2.6
SOURCE	321	1.9
REFBP	299	1.8
IMAGE	212	1.2
FLD	167	1.0
all others	1,166	6.8
Total	17,070	100.0

Table 2. Percentage of time spent in the PLAINT subroutine

PLAINT Percentage			
Number of plates	8	16	24
Percentage	38.2	49.1	57.1

Performance analysis revealed that the PLAINT subroutine is the workhorse of NEC-BSC, using up most of the execution time with calculations of intersection coordinates. Analyzing the subroutine itself to determine its complexity showed this value to be proportional to the number of objects, or $O(n)$, because it checks for a possible intersection with every other object. The intersection algorithm assumes that each object in turn extends to infinity in every direction, calculates an intersection point on the infinite object, and then determines if that intersection point lies within the bounds of the finite object. If the intersection point does not lie within the bounds of the finite object, then there is no intersection, and processing continues. If an intersection does occur, then processing for

that object stops, since the ray cannot propagate freely in the desired direction. Since some of the UTD terms have a complexity of $O(n^3)$, the overall complexity of NEC-BSC is $O(n^4)$, where n refers to the number of objects. For more detail on the functionality of NEC-BSC, see Appendix A.

5.3 Parallelization with Dynamic Load Balancing

The earlier work of Scott Suhr used a static load balancing technique to partition the work among the processors. In his work, he was limited to a small number of sample input files which were relatively simple in nature, not being truly representative of an actual problem. By analyzing the structure of the input data files, it was possible to construct more complex examples that would be more indicative of the true performance of NEC-BSC in both serial and parallel environments. These new input files, with a greater complexity than the originals, are then used in subsequent runs of the statically load balanced version of NEC-BSC in order to gather data on its efficiency as the complexity of the problem increased. The number of nodes used during these runs is also increased to investigate the effects of further scaling. Table 3 shows how the increased complexity and scaling affected the overall efficiency of the program. In the best case, efficiency increased from 23% to 80% as the number of plates increased to 32 when scaled to eight processors and still showed good efficiency (67%) when run with sixteen processors.

Table 3. Static Load Balancing Efficiency versus number of nodes (iPSC/860)

Num Nodes	Elapsed Time (msec)	Efficiency	Speedup
1	1,035,545	100.00	1.00
2	531,373	97.44	1.95
4	280,112	92.42	3.70
8	159,945	80.93	6.47
16	96,011	67.41	10.79
32	—0	—	—-
64	—0	—	—-

A modification of the standard dynamic load balancing technique introduces the ability to "hide" some of the time spent in communication as the nodes request more data, and the host responds. This method makes use of an ability inherent to the iPSC/series hypercubes. This ability arises from the way the messages are handled. The operating system that controls the operation of the nodes allows for three types of messages: synchronous (csend, crecv), asynchronous (isend, irecv), and interrupt generating messages

(hsend, hrecv). Any type of send can be processed by any type of receive command, so a send-receive pair need not be of the same type.

Furthermore, for the "csend" command, two different message protocols are used (21). If the message is less than 100 bytes in length, the message is sent along with the necessary header information to the receiving node. If sufficient buffer space is available for that message, an acknowledgement is sent back to the receiving node. Once the sending node has received the acknowledgement, it continues on with its computations. If the message is longer than 100 bytes, only the header information is sent initially, and the sending node waits until a path has been established, and the receiving node returns an initial acknowledgement. The sending node then proceeds to send the body of the message. Once the body of the message has been sent, the sending node continues with its normal execution. Before it can resume this normal operation, the sending node must wait until the message has been completely received. In this situation, the receiving node does not send the acknowledgement until it has executed a receive instruction of some type. This means that the sending node could wait while the receiving node executes some calculations, resulting in idle time.

The proposed modification to the normal dynamic load balancing technique is based on the fact that the receiving node need only notify the controller that it requires more data. Since this requires only the node number of the requester, a short message accomplishes this function. Such a short message could be sent before the node finishes processing the current data set. As the message travels to the controller, the requesting node finishes processing its current set of data. Meanwhile, the controller receives the request, determines the next set of data for that node, and sends the next set to the requesting node. By the time the requesting node completes its current data set, the next data set is already stored in a local buffer, and it can begin processing that next data set without waiting for any communications. This approach is implemented with four rays in each data set, allowing the worker nodes to request the next data set after three of the four rays have been processed.

Throughout the code development stage, standard software project management techniques are used. The dynamic load balancing is implemented incrementally, and configuration management is used to ensure that the changes do not become chaotic. This allows a quick recovery if an attempt does not succeed. The configuration management of this project is accomplished by establishing a baseline version of the program and working from that basis. Once a modification is shown to be correct in its function, a new baseline

is established. Even though there was only one person working on this project, because of the size of the code (> 20,000 lines of code), all work had to be carefully coordinated to minimize thrashing. The use of these techniques saved many hours of searching for problems!

The dynamically balanced version of NEC-BSC also shows good efficiency when scaled over eight processors. Table 4 shows its performance as a function of the number of processors, scaling up to as many as 64 processors. It is also important to note that although the overhead associated with dynamic load balancing tends to increase the overall execution time, an increase in the execution times of the nodes also tends to counterbalance this effect. It is disappointing to note that the dynamically balanced version of NEC-BSC is significantly slower than the statically balanced version. This difference in execution time is a mystery since the increased message traffic cannot account for this. In the most complex example, the dynamically balanced version sends exactly 188 more short messages than the statically load balanced version. At less than 75 microseconds per message (21), this amounts to a total time of approximately fifteen milliseconds, yet the execution times differ by as much as fifty seconds! Since both versions collect the data in much the same way and use the same method to calculate the necessary field values, the cause for this large discrepancy is unknown, and every attempt to explain it has been unsuccessful. It is also important to note that these timing values include the time required to gather the output data to the host and write the results out to disk. If those output times are removed, the actual time spent by the nodes in calculation and communication shows even more efficiency, and the effects of the internal message passing could also be measured accurately.

Table 4. Dynamic Load Balancing Efficiency versus number of nodes (iPSC/860)

Num Nodes	Elapsed Time (msec)	Efficiency	Speedup
1	2,363,036	100.00	1.00
2	1,210,133	97.63	1.95
4	632,812	93.35	3.73
8	340,853	86.66	6.93
16	199,694	73.96	11.83
32	130,300	56.67	18.14
64	100,767	36.64	23.45

A possible explanation for the increase in execution time involves the possibility that large numbers of messages can cause the system to bog down if all the messages cannot

be handled quickly enough. This phenomenon is known as "bottlenecking" and occurs when messages build up in the receiving queue of a node (10). To see if this is indeed the case, some tests were performed on one node, measuring the execution time for the same data set, but varying the number of rays calculated. Table 5 shows that the actual time to calculate a single ray is about 1.6 seconds. The elapsed times are adjusted to account for the time spent sending the final output data to the host so that only the time spent calculating the paths of the rays is be used. The round trip consists of a short message (four bytes in length) from a node to the host requesting more data, and another short message (eight bytes in length) from the host to the requesting node that contains the next data set to be calculated. A four byte message requires less than 75 microseconds to arrive (assuming no contention), while an eight byte message takes about 77 microseconds (21). Excluding the time required to calculate the values for the next data set (considered inconsequential: less than ten integer operations on a 40 MHz clock) (4), the total time for the round trip is about 150 microseconds. With 6.4 seconds between each request, then, ideally, more than 42,000 messages can be processed in each six second period. If the time at which a message is sent is a normally distributed random variable, the effects of bottlenecking would not become significant until the number of messages approached 50% of the theoretical maximum (with one message per node). Therefore, for this application, over 20,000 nodes could be handled by one controller. This means that bottlenecking even with 64 nodes cannot be the cause of the increased execution time, and this idiosynchracy remains a mystery.

Table 5. Execution Time for One Ray (iPSC/860)

Num Rays	Message Size	Elapsed Time (msec)	Adjusted Time (msec)
5	160	10,521.3	10,521.07
1440	46,080	2,325,603.0	2,325,585.28
	Average		1613.285

5.4 Summary

This chapter details the results of this thesis effort. The performance of the serial version of NEC-BSC is evaluated, and previous work in parallelizing the program is analyzed on the basis of its performance. The dynamic load balancing technique chosen in Chapter 4 is implemented and the results compared with the other two versions. In terms

of execution time, the results are disappointing when compared to the statically balanced version, but the overall improvement over the serial version is excellent. The reason for the difference in execution time is addresses as an area for future work. The next chapter discusses the conclusions of this thesis investigation and makes recommendations for future work.

VI. Conclusions and Recommendations

6.1 Introduction

This chapter contains the conclusions and realizations reached through this thesis effort. The results of the parallelization effort are discussed, and future possibilities for work are presented.

6.2 Conclusions

It is both possible and profitable to convert existing serial programs to a parallel application if the resulting product is to be used on a regular basis. The resulting savings in execution time can rapidly recoup the time spent in development. For example, if NEC-BSC is to be regularly used with data sets of 32 objects, Table 4 shows a savings of approximately 38 minutes per run (over one node) on a 64 node machine. This translates into a savings of about 160 minutes (per run) versus a VAX 11/780. This savings is also just over 37 minutes for 32 nodes (versus one node) or 159 minutes versus the VAX 11/780. If the program is executed twice a day (on a 32 object data set), then a savings of one hour per day (versus one node or five hours versus a VAX 11/780) can be realized. This translates into five hours a week, or twenty hours a month (or 25 hours a week and 100 hours a month versus the VAX 11/780). Within a short time, the development effort can be recouped, and productivity will increase. Although the execution time on 64 nodes experienced a dramatic decrease in efficiency (compared to the execution time on 32 nodes), with even more complex data sets, this efficiency rises. This trend is supported by the data indicating that the amount of calculation performed by the program increases greatly as the complexity of the number of objects in the target geometry increases. More calculations means that the ratio of the calculations performed to communications sent and received increases. Since the grain of the iPSC/860 remains constant, more calculations per communication increases the efficiency of the application.

6.3 Recommendations

NEC-BSC itself could benefit greatly from a thorough examination with standard software engineering techniques. The modules of the program are both tightly coupled and have loose cohesion. The efficiency of the program itself could be increased greatly if a project were undertaken to improve the code. Such an examination would require

more than one person, and every engineer would need to be familiar with the theories of electric and magnetic fields as they are propagated through space. Another area for work would be to convert the source code from FORTRAN to another language that supports dynamic allocation of memory. FORTRAN wastes large amounts of space when the full range of a program's capabilities are not used. Using a computer language such as C or Ada would allow memory to be used more efficiently. Additionally, the capabilities of NEC-BSC need to be expanded. The program is currently limited to 36 objects, and 1801 rays. Although the number of rays is probably sufficient for good accuracy, complex objects cannot be accurately modeled with a small number of objects, and today's computers have the necessary memory available to expand these limits. Assuming 512 bytes of storage for each object, 2,000 objects could be stored in 1M of memory. The 1800 possible rays would take up only 100K of memory, so a machine with 2Mbytes or more of memory could easily handle complex scenes modeled with a large number of primitive objects. Computer languages such as C or Ada also manage available memory much better than FORTRAN, and this efficiency could be used when converting NEC-BSC to one of these languages.

Another area for work lies in increasing the number of UTD terms that are handled. The currently supported terms handle up to three reflections and two diffractions. A higher number of reflections would be desirable, and could be achieved with no loss of efficiency if a new algorithm were developed to take advantage of the available memory, using it to reduce redundant calculations. A possible approach to this would be to compare a simple value (calculated for each UTD term and object) with a desired value. If the calculated values are stored in a large matrix, then they can be easily referenced, instead of recalculating intersection points.

A strong possibility for future work involves re-engineering the problem based on the functionality of NEC-BSC. If done with a parallel environment in mind, the resulting application could be considerably more efficient than previous parallelization efforts. During such work, UNITY would be a strong asset in designing the program, allowing the application to be tailored to the architectural characteristics of the target machine. One reason that such an approach would be successful is that NEC-BSC was originally written for a small-memory model architecture. Today's machines all possess a great deal more memory than previous versions, and in order to make use of the large memory sizes currently available, it may be necessary to completely rewrite the main sections of the program. The smaller subroutines which perform basic scientific calculations could probably be brought over unchanged (18).

NEC-BSC could also be ported to a different computer architecture to investigate the effects that a different machine structure would have on efficiency. It is possible that a different decomposition for a different target machine could result in better results. Conversely, it is also possible that a different architecture would not be as efficient. Possibilities for such a conversion are the Connection Machine, and the iWARP project. Both these machines are fine-grain architectures (10), and a distribution of the objects across these machines is certainly possible without a great loss of efficiency. The characteristics of the Connection Machine and the iWARP project differ from those of the iPSC/860, and a conversion to either of these machines could take advantage of their strengths. The new Intel machine, the Paragon, although based on the same microprocessor as the iPSC/860, features greatly improved communications, reducing its grain size. This factor could also increase the efficiency of an application such as NEC-BSC.

Appendix A. *NEC-BSC*

A.1 Introduction

This appendix gives a brief description of the functionality of NEC-BSC and some samples of the parallel code that allowed the program to run on the iPSC/860 hypercube.

A.2 Functions

NEC-BSC provides the ability to calculate the simulated radar cross-section (RCS) of an object when incident electromagnetic energy encounters that object. NEC-BSC also allows the researcher to calculate the strength of the simulated electric and magnetic fields at any point within the three-dimensional reference coordinate system of the object. The effects of antennas and dipoles are accurately represented, and objects may be made of various types of materials including perfectly conducting metallic, transparent thin dialectric, or a dialectric covered surface. An object may be composed of multiple sections, each of a fundamental type. Fundamental types include flat plates and cylinders (including elliptical cylinders), and provision is made for a ground plane. Supported functions include far and near zone patterns and back or bistatic scattering. Parameters that influence the calculations include number of sources, number of receivers, presence of antennas, and varying frequencies. The program is written in FORTRAN 77, and comprises approximately 20,000 lines of code, including comments. Figures 8 and 9 show a simple scene geometry from the top and side views, and Figure 10 shows the output from NEC-BSC for the scene shown in Figures 8 and 9.

A.3 Samples of the Parallel Code

The full source code may be found in the ~pwork/necbsc/dyn_bsc subdirectory on mbvsrm.mbvlab.wpafb.af.mil. The first example is from the host program and contains the declarations and code necessary to control the node programs.

A.3.1 Samples of the dynamic load balancer

```
      Program run_bscnode
CCC

c---  Define variables for use as message types
```

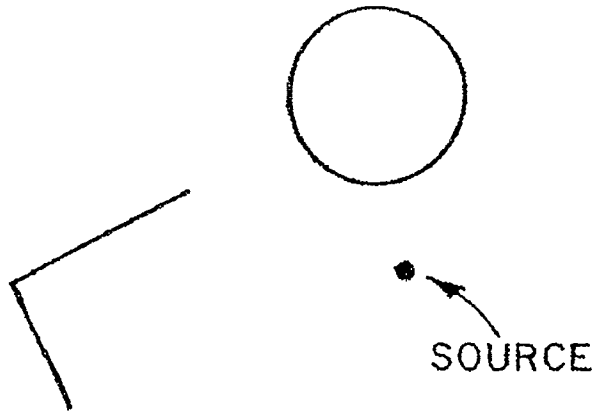

Figure 8. Top view of a sample scene geometry

```
      integer WORK_TYPE, RAY_TYPE
      integer NEW_DATA (2), NEW_SIZE
      integer NEXT

c---  Initialization of the message size variable

      WORK_TYPE = 222
      RAY_TYPE = 333
      NEW_SIZE = 8

c---  Initialization of the starting point for the dynamic
c---       balancing

      NEW_DATA (1) = NUM_NODES * 4 + 1
      NEW_DATA (2) = NEW_DATA (1) + 3
      NEXT = NEW_DATA (2) + 1

c---  Check for a smaller data set than the nodes can process

      if (NEW_DATA (1).GT.NPN) then
          write (*, *) 'Input data too small for the current cube
     2                  size.  Please try a'
          write (*, *) 'smaller cube or a larger data set'

          STOP
```

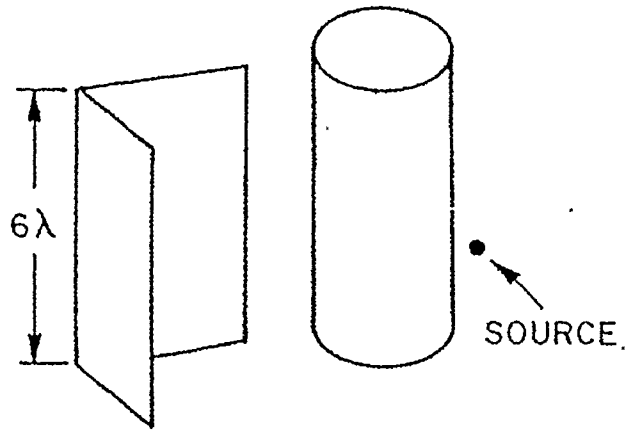

Figure 9. Side view of a sample scene geometry

```
      end if

c---  Check for less than four rays in the next data set

        if (NEXT.GT.NPN) NEXT = 0
        if (NEW_DATA (2).GT.NPN) NEW_DATA (2) = NPN

c---  If there is a current set of rays to be processed, proceed

1190    if (NEW_DATA (2).GT.0) then

c---  Receive a request for more data from a node and send out the
c---       next set to be processed

            call crecv (WORK_TYPE, NODE_NUM, INT_SIZE)
            call csend (RAY_TYPE, NEW_DATA, NEW_SIZE, NODE_NUM, 0)

c---  Calculate the next set of data for processing.  If finished,
c---       set the next ray to 0, so the nodes upon receipt of
c---       a zero will stop processing.

            if (NEXT.GT.0) then
              NEW_DATA (1) = NEXT
              if (NEXT + 4.GT.NPN) then
                NEW_DATA (2) = NPN
                NEXT = 0
              else
```

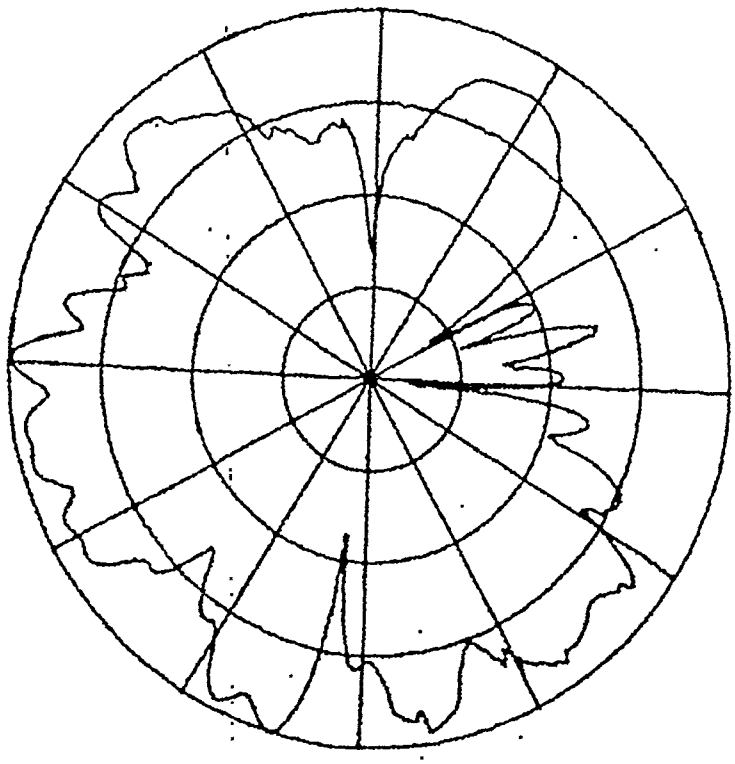

Figure 10. Far zone results for the sample scene geometry

```
          NEW_DATA.(2) = NEXT + 3
          NEXT = NEW_DATA (2) + 1
        end if
      else
        if (NEW_DATA (1).GT.0) then
          NEW_DATA.(1) = 0
        else
          if (NEW_DATA (2).GT.0) then
            NEW_DATA (2) = 0
          end if
        end if
      end if

c---  Return to the top of the loop for another pass

        goto 1190
      end if

c---  Tell the remaining nodes to stop processing
```

```
        if (NUM_NODES.GT.1) then
          do 444 I = 1, NUM_NODES
            if (I.NE.NODE_NUM + 1) then
              call csend (RAY_TYPE, NEW_DATA, NEW_SIZE, I - 1, 0)
            end if
444       continue
        end if
```

A.3.2 Samples of the node program This section contains excerpts from the node program. These excerpts contain the declarations and code necessary to allow the node program to communicate with the host program, sending requests for more work, and processing the next data set after it arrives.

```
        PROGR.. : NECBSC

C!!!    NEC-BSC Version 3.2   ( Updated 26-OCT-89 )
C!!!    iPSC mod # :    4.0   ( Updated  2 May 91 )
C!!!    iPSC mod # :    5.0   ( Updated 15 Aug 91 )

                .
                .
                .

c---   variables for saving the output data in a different
c---                    format

        COMPLEX CT1(NOX),ET1(3,NOX),HT1(3,NOX)
        integer RAYS(1801)

c---  Add the necessary declarations to allow variable
c---    stop and start points for the pattern point loop

      integer START_POINT, NUM_LOOPS

c---  Message passing variables

      integer WORK_TYPE, RAY_TYPE

c---  Message passing variables

      integer NEW_DATA(2), NEW_SIZE

c---  Add an integer counter to count the number of rays that
c---            were processed

      integer RAY_COUNT

c---  Change 3:  Initialize the message passing variables

      WORK_TYPE = 222
      RAY_TYPE = 333
```

```
c---  Change 4:  Initialize the message passing variables

      NEW_SIZE = 8

c---  Change 9:  Initialize the ray counter

      RAY_COUNT = 0

C!!!
C!!!  3.  MAIN COMPUTATION SECTION
C!!!
C!!!  Loop thru volumetric pattern points.
        DO 1190 IIV=1,NPV
              .
              .
              .

c---  Initialize the loop variables

        START_POINT = MY_NODE * 4 + 1
        NPNP = START_POINT + 3

c---   Start the pattern point loop

        DO 1100 IIC = START_POINT, NPNP
             II=IIC

            .
            .
            .

c---  Change 3:  Send message to host requesting more data

            if (NPNP - START_POINT.EQ.3) then
              if (IIC.EQ.START_POINT + 2) then
                call csend (WORK_TYPE, MY_NODE, INT_SIZE, MY_HOST, 81)
              end if
            else
              if (START_POINT.EQ.NPNP) then
                call csend (WORK_TYPE, MY_NODE, INT_SIZE, MY_HOST, 81)
              else
                if (IIC.EQ.NPNP - 1) then
                  call csend(WORK_TYPE, MY_NODE, INT_SIZE,
     2                              MY_HOST, 81)
```

```
                  end if
                end if
              end if

c---  Add a counter to determine exactly how many rays were processed

              RAY_COUNT = RAY_COUNT + 1

c---  Ending place for the pattern point loop

1100        CONTINUE

c---  Receive the next set of poiints from the host

            call crecv (RAY_TYPE, NEW_DATA, NEW_SIZE)
            START_POINT = NEW_DATA (1)
            NPNP = NEW_DATA (2)

            if (START_POINT.GT.0) goto 1180
            NPNP = RAY_COUNT

C!!!  End of volumetric pattern loop.
1190    CONTINUE

        END
```

Appendix B. *Sample output of NEC-BSC*

B.1 Sample timing information from NEC-BSC

Following is a sample of the information printed to the screen during execution of the parallel version of NEC-BSC. The individual node timings as well as the time required to gather the data from the nodes and write the results to disk are given. Finally, the total execution time is listed. During each run of parallel NEC-BSC, the number of nodes assigned does not change. Each new run displays how many nodes are being used, and the corresponding times for that run.

```
Running NEC-BSC:

Number of nodes attached:          64

Input Filename = "ex6g1.inp"

Elapsed Time:

   Node       Total Time (msec)

     0             65288
     1             68037
     2             65355
     3             63951
           .
           .
           .
    61             63704
    62             63986
    63             68287

Output elapsed time (node 0, msec):     18356

Host CPU time required for startup:         0
Host CPU time required for output:      22820

Total Host CPU time required:           32640
Approx. total elapsed time required:    100767
     (max node + Startup + Output)
```

```
Running NEC-BSC:

Number of nodes attached:          32

Input Filename = "ex6g1.inp"

Elapsed Time:

   Node       Total Time (msec)

     0          97,777
     1          97,167
     2          98,330
     3          97,119
           .
           .
           .
    29          98,265
    30          97,984
    31          98,070

Output elapsed time (node 0, msec):      17,795

Host CPU time required for startup:           0
Host CPU time required for output:       22,460

Total Host CPU time required:            32,100
Approx. total elapsed time required:    130,300
     (max node + Startup + Output)

           .
           .
           .

Running NEC-BSC:

Number of nodes attached:           2

Input Filename = "ex6g1.inp"

Elapsed Time:

   Ncde       Total Time (msec)

     0          1,169,889
```

```
     1          1,175,933

Output elapsed time (node 0, msec):      11,399

Host CPU time required for startup:           0
Host CPU time required for output:       24,830

Total Host CPU time required:            34,220
Approx. total elapsed time required:  1,210,133
     (max node + Startup + Output)



Running NEC-BSC:

Number of nodes attached:           1

Input Filename = "ex6g1.inp"

Elapsed Time:

   Node       Total Time (msec)

     0          2,327,646

Output elapsed time (node 0, msec):      10,177

Host CPU time required for startup:           0
Host CPU time required for output:       25,700

Total Host CPU time required:            35,400
Approx. total elapsed time required:  2,363,036
     (max node + Startup + Output)
```

B.2 Sample program output

Following are portions of an output file produced by NEC-BSC for the input file ex6g1.inp. The essential data of the input file is repeated, and then the electric field strength values for each of the specified angles is given. After the electric field strength values come the total field strength values computed by the program.

```
**************************************************************************
*                                                                        *
*  NEC-BSC      3.2i5.0,  12 Aug 91                                      *
*                                                                        *
*     The Ohio State University                                          *
*     Electroscience Laboratory                                          *
*     1320 Kinnear Rd.                                                   *
*     Columbus, Ohio 43212                                               *
*                                                                        *
*     Written by Ronald J. Marhefka                                      *
*                                                                        *
*     Modified for the Intel iPSC/2 & iPSC/860 by                        *
*     Paul R Work, Scott Suhr and                                        *
*     Dr Gary B. Lamont                                                  *
*     Air Force Institute of Technology                                  *
*     AFIT/ENG                                                           *
*     Wright Patterson AFB, OH 45433-6583                                *
*                                                                        *
**************************************************************************

**************************************************************************
*                                                                        *
*  CE: TWO EIGHT SIDED BOXES TEST, EACH WITH EIGHT OUTLYING PLATES       *
*                  EX 6D1.                                               *
*                                                                        *
*                                                                        *
**************************************************************************

**************************************************************************
*                                                                        *
*  US:                                                                   *
*                                                                        *
*                                                                        *
*    SOURCE LENGTH HS AND WIDTH HAWS ARE ASSUMED TO BE IN WAVELENGTHS    *
*                                                                        *
**************************************************************************
```

```
*****************************************************************************
*                                                                           *
*  FR:                                                                      *
*                                                                           *
*                                                                           *
*     FREQUENCY=   9.940 GIGAHERTZ                                          *
*                                                                           *
*     WAVELENGTH=   0.030160 METERS                                         *
*                                                                           *
*****************************************************************************

*****************************************************************************
*                                                                           *
*  PF:                                                                      *
*                                                                           *
*                                                                           *
*     PATTERN AXES ARE AS FOLLOWS:                                          *
*                                                                           *
*   VPC(1,1)=  1.00000  VPC(1,2)=  0.00000  VPC(1,3)=  0.00000              *
*                                                                           *
*   VPC(2,1)=  0.00000  VPC(2,2)=  1.00000  VPC(2,3)=  0.00000              *
*                                                                           *
*   VPC(3,1)=  0.00000  VPC(3,2)=  0.00000  VPC(3,3)=  1.00000              *
*                                                                           *
*     PHI IS BEING VARIED WITH THETA=   90.00000                            *
*                                                                           *
*     START=    0.00000 STEP=    0.50000 NUMBER=  721                       *
*                                                                           *
*****************************************************************************

*****************************************************************************
*                                                                           *
*  SG:                                                                      *
*                                                                           *
*                                                                           *
*     THIS IS SOURCE NO.   1 IN THIS COMPUTATION.                           *
*                                                                           *
*                                                                           *
*     THIS IS AN ELECTRIC SOURCE OF TYPE  -2                                *
*                                                                           *
*     SOURCE LENGTH=   0.50000 AND WIDTH=   0.00000 WAVELENGTHS             *
*                                                                           *
*     SOURCE LENGTH=   0.01508 AND WIDTH=   0.00000 METERS                  *
*                                                                           *
```

```
*     THE SOURCE WEIGHT HAS MAGNITUDE=   1.00000 AND PHASE=   0.00000   *
*                                                                        *
*                                                                        *
*   SOURCE#    INPUT LOCATION IN METERS     ACTUAL LOCATION IN METERS    *
*   -------   --------------------------   --------------------------   *
*                                                                        *
*       1     -20.120,   0.000,   0.000    -20.120,   0.000,   0.000    *
*                                                                        *
*                                                                        *
*     THE FOLLOWING SOURCE ALIGNMENT IS USED:                            *
*                                                                        *
*   VXSS(1,1, 1)= 1.00000  VXSS(1,2, 1)= 0.00000  VXSS(1,3, 1)= 0.0000  *
*                                                                        *
*   VXSS(2,1, 1)= 0.00000  VXSS(2,2, 1)= 1.00000  VXSS(2,3, 1)= 0.0000  *
*                                                                        *
*   VXSS(3,1, 1)= 0.00000  VXSS(3,2, 1)= 0.00000  VXSS(3,3, 1)= 1.0000  *
*                                                                        *
**************************************************************************

**************************************************************************
*                                                                        *
*  PG: FRONT                                                             *
*                                                                        *
*                                                                        *
*     THIS IS PLATE NO.    1 IN THIS SIMULATION.                         *
*                                                                        *
*                                                                        *
*     METAL PLATE USED IN THIS SIMULATION                                *
*                                                                        *
* PLATE#  CORNER#    LOCATION IN METERS   ACTUAL LOCATION IN METERS      *
* ------  -------  --------------------  ------------------------      *
*                                                                        *
*   1       1      0.122,  0.102, -0.100   0.122,   0.102,  -0.100      *
*                                                                        *
*   1       2      0.122,  0.102,  0.100   0.122,   0.102,   0.100      *
*                                                                        *
*   1       3      0.122, -0.102,  0.100   0.122,  -0.102,   0.100      *
*                                                                        *
*   1       4      0.122, -0.102, -0.100   0.122,  -0.102,  -0.100      *
*                                                                        *
**************************************************************************

**************************************************************************
*                                                                        *
*  PG: FAR FRONT                                                         *
```

```
*                                                                              *
*                                                                              *
*     THIS IS PLATE NO.    2 IN THIS SIMULATION.                               *
*                                                                              *
*                                                                              *
*     METAL PLATE USED IN THIS SIMULATION                                      *
*                                                                              *
* PLATE#  CORNER#    LOCATION IN METERS   ACTUAL LOCATION IN METERS            *
* ------  -------  ---------------------  -------------------------           *
*                                                                              *
*   2       1      2.122,  0.102, -0.100   2.122,   0.102,  -0.100             *
*                                                                              *
*   2       2      2.122,  0.102,  0.100   2.122,   0.102,   0.100             *
*                                                                              *
*   2       3      2.122, -0.102,  0.100   2.122,  -0.102,   0.100             *
*                                                                              *
*   2       4      2.122, -0.102, -0.100   2.122,  -0.102,  -0.100             *
*                                                                              *
********************************************************************************

********************************************************************************
*                                                                              *
*  PG: FRONT BOX 2                                                             *
*                                                                              *
*                                                                              *
*     THIS IS PLATE NO.    3 IN THIS SIMULATION.                               *
*                                                                              *
*                                                                              *
*     METAL PLATE USED IN THIS SIMULATION                                      *
*                                                                              *
* PLATE#  CORNER#    LOCATION IN METERS    ACTUAL LOCATION IN METERS           *
* ------  -------  ---------------------  -------------------------           *
*                                                                              *
*   3       1      10.122,  0.102, -0.100   10.122,   0.102,  -0.100           *
*                                                                              *
*   3       2      10.122,  0.102,  0.100   10.122,   0.102,   0.100           *
*                                                                              *
*   3       3      10.122, -0.102,  0.100   10.122,  -0.102,   0.100           *
*                                                                              *
*   3       4      10.122, -0.102, -0.100   10.122,  -0.102,  -0.100           *
*                                                                              *
********************************************************************************
```

.
.

```
*************************************************************************
*                                                                       *
*  PG: FAR BOTTOM BOX 2                                                 *
*                                                                       *
*                                                                       *
*     THIS IS PLATE NO.   32 IN THIS SIMULATION.                        *
*                                                                       *
*                                                                       *
*     METAL PLATE USED IN THIS SIMULATION                               *
*                                                                       *
* PLATE#  CORNER#    LOCATION IN METERS    ACTUAL LOCATION IN METERS    *
* ------  -------  ----------------------  -------------------------    *
*                                                                       *
*  32       1      10.000,  0.171, -2.100   10.000,   0.171,  -2.100    *
*                                                                       *
*  32       2      10.122,  0.102, -2.100   10.122,   0.102,  -2.100    *
*                                                                       *
*  32       3      10.122, -0.102, -2.100   10.122,  -0.102,  -2.100    *
*                                                                       *
*  32       4      10.000, -0.171, -2.100   10.000,  -0.171,  -2.100    *
*                                                                       *
*  32       5       9.878, -0.102, -2.100    9.878,  -0.102,  -2.100    *
*                                                                       *
*  32       6       9.878,  0.102, -2.100    9.878,   0.102,  -2.100    *
*                                                                       *
*************************************************************************
```

THE FAR ZONE ELECTRIC FIELD

THE FIELDS ARE REFERENCED TO THE PATTERN COORDINATE SYSTEM

		E-THETA		
THETA	PHI	MAGNITUDE	PHASE	DB
90.00	0.00	2.1319E+01	42.52	-2.20
90.00	0.50	5.7678E+01	59.92	6.45
90.00	1.00	5.3923E+01	87.97	5.86
90.00	1.50	5.8013E+01	134.44	6.50

90.00	358.50	5.8006E+01	134.45	6.50
90.00	359.00	5.3926E+01	87.96	5.86
90.00	359.50	5.7659E+01	59.92	6.44
90.00	360.00	2.2964E+01	44.18	-1.55

****

TOTAL RADIATION INTENSITY IN DB

THE FIELDS ARE REFERENCED TO THE PATTERN COORDINATE SYSTEM

THETA	PHI	MAJOR	MINOR	TOTAL	AXIAL RATIO	TILT ANG	SENSE
90.00	0.00	-2.20	-100.00	-2.20	0.00000	0.00	LIN
90.00	0.50	6.45	-92.89	6.45	0.00001	0.00	RT
90.00	1.00	5.86	-100.00	5.86	0.00000	0.00	LIN
90.00	1.50	6.50	-100.00	6.50	0.00000	0.00	LIN
90.00	358.50	6.50	-100.00	6.50	0.00000	0.00	LIN
90.00	359.00	5.86	-100.00	5.86	0.00000	0.00	LIN
90.00	359.50	6.44	-87.83	6.44	0.00002	0.00	LFT
90.00	360.00	-1.55	-100.00	-1.55	0.00001	0.00	LIN

****

General

The Finite Difference Time Domain method is a discretization of the Maxwell Equations in differential form (curl equations). Starting with Maxwell's equations:

$$\nabla \times H = \varepsilon \frac{\partial E}{\partial t} + \sigma_e E \tag{6}$$

$$\nabla \times E = -\mu \frac{\partial H}{\partial t} - \sigma_m H \tag{7}$$

where μ is the magnetic permeability, ε is the dielectric permittivity, σ_e is the total equivalent conductivity giving rise to electric dissipative currents, and σ_m is the corresponding parameter giving rise to magnetic dissipative currents. All parameters are real. These equations are separated according to their vector components into a scalar form:

$$\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} = \varepsilon \frac{\partial E_x}{\partial t} + \sigma_e E_x \tag{8}$$

$$\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} = \varepsilon \frac{\partial E_y}{\partial t} + \sigma_e E_y \tag{9}$$

$$\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} = \varepsilon \frac{\partial E_z}{\partial t} + \sigma_e E_z \tag{10}$$

$$\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z} = -\mu \frac{\partial H_x}{\partial t} - \sigma_m H_x \tag{11}$$

$$\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x} = -\mu \frac{\partial H_y}{\partial t} - \sigma_m H_y \tag{12}$$

$$\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y} = -\mu \frac{\partial H_z}{\partial t} - \sigma_m H_z \tag{13}$$

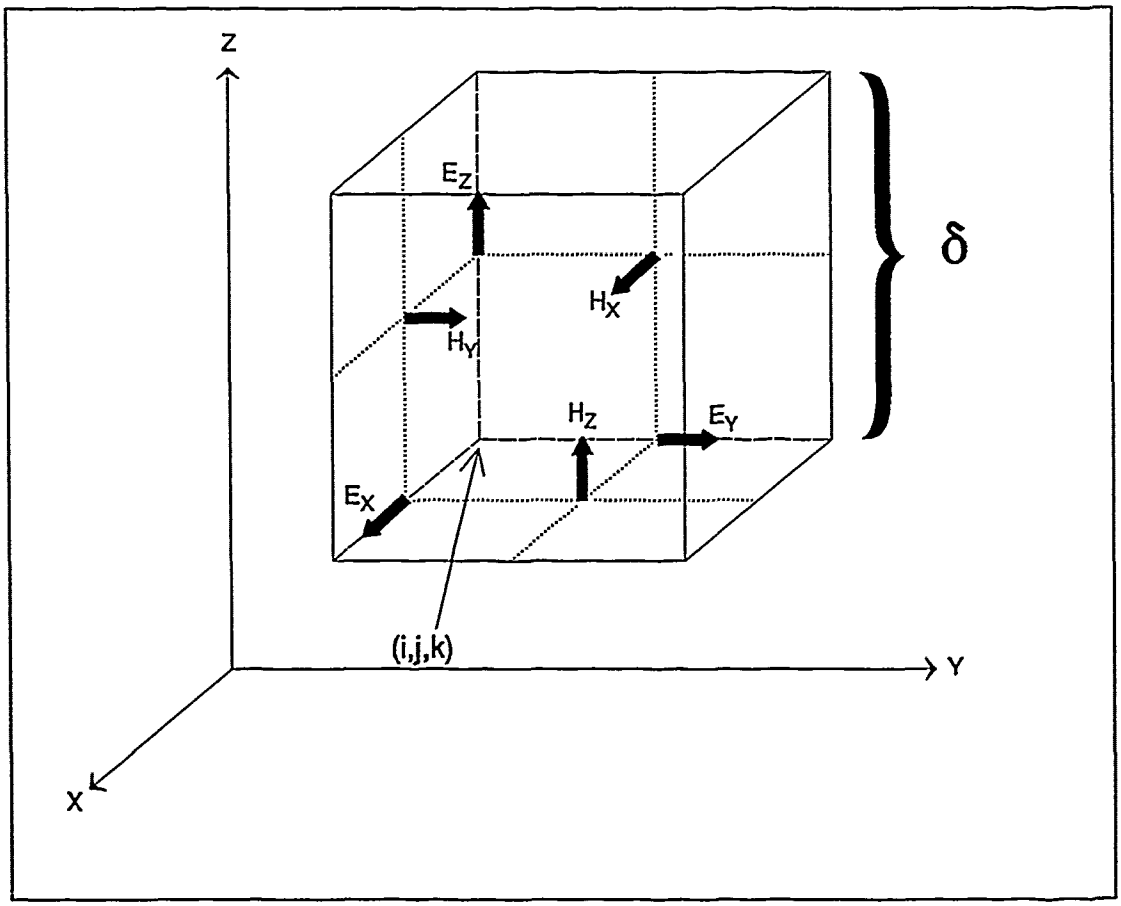

Figure 11 -- Yee Cell

The FDTD method uses centered differences which are based on the following first-order approximations to the derivative:

$$\frac{\partial F(i,j,k,t)}{\partial x} = \frac{F(i+\frac{\delta x}{2},j,k,t) - F(i-\frac{\delta x}{2},j,k,t)}{\delta x} + O(\delta x^2) \tag{14}$$

$$\frac{\partial F(i,j,k,t)}{\partial t} = \frac{F(i,j,k,t+\frac{\Delta t}{2}) - F(i,j,k,t-\frac{\Delta t}{2})}{\Delta t} + O(\Delta t^2) \tag{15}$$

The derivatives in space and time in Maxwell's equations are replaced by these centered differences. Evaluation of the values of E and H fields are offset in space

by one half intervals as shown in Figure 11. Notice that the H field values are defined as entering the cell and the E field values are defined along the three orthogonal edges nearest to the origin (indexes i,j,k are positive valued) and, in this study,

$$\delta=\delta x=\delta y=\delta z \tag{16}$$

E and H are also offset in time by one half intervals. The FDTD method solves alternately for E and H as time is incremented in one half time steps. The individual equations are as follows:

$$E_x^{n+1}(i+\tfrac{1}{2},j,k)=\frac{1-\dfrac{\sigma_e(i+\frac{1}{2},j,k)}{2\varepsilon(i+\frac{1}{2},j,k)}}{1+\dfrac{\sigma_e(i+\frac{1}{2},j,k)}{2\varepsilon(i+\frac{1}{2},j,k)}}E_x^{n}(i+\tfrac{1}{2},j,k)+\frac{\Delta t}{\varepsilon(i+\frac{1}{2},j,k)\delta}*\frac{1}{1+\dfrac{\sigma_e(i+\frac{1}{2},j,k)}{2\varepsilon(i+\frac{1}{2},j,k)}}*\begin{bmatrix} H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k)-H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j-\frac{1}{2},k) \\ +H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k-\frac{1}{2})-H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) \end{bmatrix} \tag{17}$$

$$E_y^{n+1}(i,j+\tfrac{1}{2},k)=\frac{1-\dfrac{\sigma_e(i,j+\frac{1}{2},k)}{2\varepsilon(i,j+\frac{1}{2},k)}}{1+\dfrac{\sigma_e(i,j+\frac{1}{2},k)}{2\varepsilon(i,j+\frac{1}{2},k)}}E_y^{n}(i,j+\tfrac{1}{2},k)+\frac{\Delta t}{\varepsilon(i,j+\frac{1}{2},k)\delta}*\frac{1}{1+\dfrac{\sigma_e(i,j+\frac{1}{2},k)}{2\varepsilon(i,j+\frac{1}{2},k)}}*\begin{bmatrix} H_x^{n+\frac{1}{2}}(i,j+\frac{1}{2},k+\frac{1}{2})-H_x^{n+\frac{1}{2}}(i,j+\frac{1}{2},k-\frac{1}{2}) \\ +H_z^{n+\frac{1}{2}}(i-\frac{1}{2},j+\frac{1}{2},k)-H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) \end{bmatrix} \tag{18}$$

$$E_z^{n+1}(i,j,k+1/2)=\frac{1-\dfrac{\sigma_e(i,j,k+1/2)}{2\varepsilon(i,j,k+1/2)}}{1+\dfrac{\sigma_e(i,j,k+1/2)}{2\varepsilon(i,j,k+1/2)}}E_z^{n}(i,j,k+1/2)$$
$$+\frac{\Delta t}{\varepsilon(i,j,k+1/2)\delta}*\frac{1}{1+\dfrac{\sigma_e(i,j,k+1/2)}{2\varepsilon(i,j,k+1/2)}} \tag{19}$$
$$*\begin{bmatrix} H_y^{n+1/2}(i+1/2,j,k+1/2)-H_y^{n+1/2}(i-1/2,j,k+1/2) \\ +H_x^{n+1/2}(i,j-1/2,k+1/2)-H_x^{n+1/2}(i,j+1/2,k+1/2) \end{bmatrix}$$

$$H_x^{n+1/2}(i,j+1/2,k+1/2)=\frac{1-\dfrac{\sigma_m(i,j+1/2,k+1/2)\Delta t}{2\mu(i,j+1/2,k+1/2)}}{1+\dfrac{\sigma_m(i,j+1/2,k+1/2)\Delta t}{2\mu(i,j+1/2,k+1/2)}}H_x^{n-1/2}(i,j+1/2,k+1/2)$$
$$+\frac{\Delta t}{\mu(i,j+1/2,k+1/2)\delta}*\frac{1}{1+\dfrac{\sigma_m(i,j+1/2,k+1/2)\Delta t}{2\mu(i,j+1/2,k+1/2)}} \tag{20}$$
$$*\begin{bmatrix} E_y^{n}(i,j+1/2,k+1)-E_y^{n}(i,j+1/2,k) \\ +E_z^{n}(i,j,k+1/2) \quad -E_z^{n}(i,j+1,k+1/2) \end{bmatrix}$$

$$H_y^{n+1/2}(i+1/2,j,k+1/2)=\frac{1-\dfrac{\sigma_m(i+1/2,j,k+1/2)\Delta t}{2\mu(i+1/2,j,k+1/2)}}{1+\dfrac{\sigma_m(i+1/2,j,k+1/2)\Delta t}{2\mu(i+1/2,j,k+1/2)}}H_y^{n-1/2}(i+1/2,j,k+1/2)$$
$$+\frac{\Delta t}{\mu(i+1/2,j,k+1/2)\delta}*\frac{1}{1+\dfrac{\sigma_m(i+1/2,j,k+1/2)\Delta t}{2\mu(i+1/2,j,k+1/2)}} \tag{21}$$
$$*\begin{bmatrix} E_z^{n}(i+1,j,k+1/2)-E_z^{n}(i,j,k+1/2) \\ +E_x^{n}(i+1/2,j,k) \quad -E_x^{n}(i+1/2,j,k+1) \end{bmatrix}$$

$$H_z^{n+1/2}(i+1/2,j+1/2,k)=\frac{1-\dfrac{\sigma_m(i+1/2,j+1/2,k)}{2\mu(i+1/2,j+1/2,k)}}{1+\dfrac{\sigma_m(i+1/2,j+1/2,k)}{2\mu(i+1/2,j+1/2,k)}}H_z^{n-1/2}(i+1/2,j+1/2,k)$$
$$+\frac{\Delta t}{\mu(i+1/2,j+1/2,k)\delta}*\frac{1}{1+\dfrac{\sigma_m(i+1/2,j+1/2,k)}{2\mu(i+1/2,j+1/2,k)}} \tag{22}$$
$$*\begin{bmatrix} E_x^n(i+1/2,j+1,k)-E_x^n(i+1/2,j,k) \\ +E_y^n(i,j+1/2,k) \quad -E_y^n(i+1,j+1/2,k)\end{bmatrix}$$

where δ is the lattice spacing increment, Δt is the time step increment. In order to guarantee stability, the choice of time step and spacing increments should satisfy the following:

$$v_{max}\times\Delta t\le\left(\frac{1}{\delta x^2}+\frac{1}{\delta y^2}+\frac{1}{\delta z^2}\right)^{-\frac{1}{2}} \tag{23}$$

or, in our case,

$$\Delta t\le\frac{\delta}{\sqrt{3}v_{max}} \tag{24}$$

where v_{max} is the maximum phase velocity within the computational domain. As presented, these equations can handle isotropic, inhomogeneous, lossy magnetic and lossy dielectric materials.

Note that these equations can all be represented in the following form (see Figure 12):

$$Field_{next}=Field_{prev}*K1+K2*\begin{bmatrix} Dual_1-Dual_2 \\ +Dual_3-Dual_4\end{bmatrix} \tag{25}$$

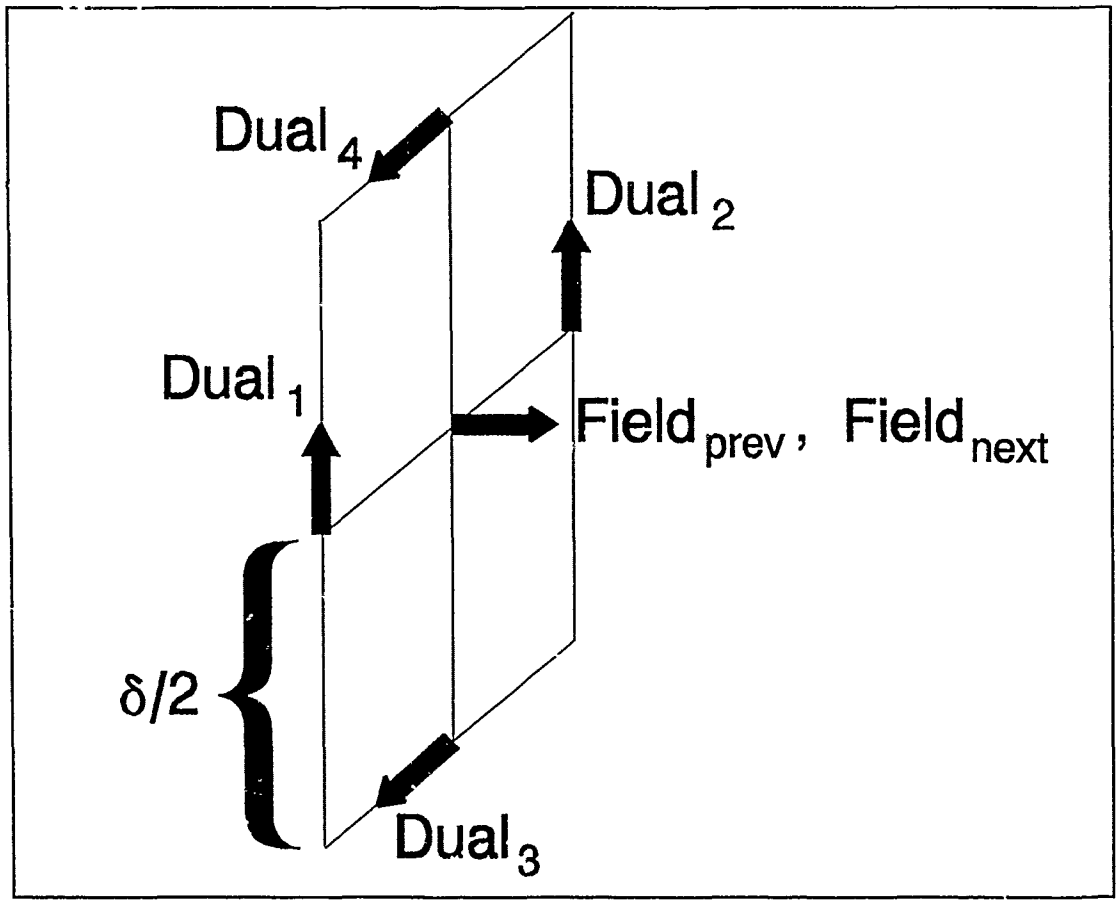

Figure 12 -- Modified Field Names

where

$$K1(i,j,k)=\frac{1-\frac{\sigma_e(i,j,k)}{2\varepsilon(i,j,k)}}{1+\frac{\sigma_e(i,j,k)}{2\varepsilon(i,j,k)}} \tag{26}$$

$$K2(i,j,k)=\frac{\Delta t}{\varepsilon(i,j,k)\delta}*\frac{1}{1+\frac{\sigma_e(i,j,k)}{2\varepsilon(i,j,k)}} \tag{27}$$

for equations (12)-(14) and

$$K1(i,j,k)=\frac{1-\frac{\sigma_m(i,j,k)}{2\mu(i,j,k)}}{1+\frac{\sigma_m(i,j,k)}{2\mu(i,j,k)}} \tag{28}$$

$$K2(i,j,k)=\frac{\Delta t}{\mu(i,j,k)\delta}*\frac{1}{1+\frac{\sigma_m(i,j,k)}{2\mu(i,j,k)}} \tag{29}$$

for equations (15)-(17), and *Dual* is the dual of the field being calculated. This simplified form leads to a straightforward method to compute these fields in hardware.

Radiation Boundary Conditions

Another computational problem area of the FDTD method is the radiation boundary condition that must be satisfied at all six faces of the volume. It arises from the fact that the fields are supposedly in an unbounded space, yet researchers lack the computational power and time to even approximate this environment. Therefore, the cell lattice is truncated along planes close to the subject of study and a radiation boundary condition is imposed. This condition attempts to determine values for the fields lying on the external boundary, since there are no fields external to these with which to calculate them using the standard cell equations. Although not nearly as computationally intense as the $O(n^3)$ FDTD cell equations problem, the calculation time for these exterior points increases as $O(n^2)$, where n is the linear dimension of the problem space. In large problems, this may account for a significant amount of time.

Many researchers using the FDTD method employ the second-order Mur radiation boundary equation, which, for the x=0 face, is:

$$
\begin{aligned}
E_z^{n+1}(0,j,k+\tfrac{1}{2}) = & -E_z^{n-1}(1,j,k+\tfrac{1}{2}) \\
& + \frac{c\Delta t-\delta}{c\Delta t+\delta} * \{E_z^{n+1}(1,j,k+\tfrac{1}{2}) + E_z^{n-1}(0,j,k+\tfrac{1}{2})\} \\
& + \frac{2\delta}{c\Delta t+\delta} * \{E_z^{n}(0,j,k+\tfrac{1}{2}) + E_z^{n}(1,j,k+\tfrac{1}{2})\} \\
& + \frac{(c\Delta t)^2}{2\delta(c\Delta t+\delta)} \\
& * \begin{bmatrix}
E_z^{n}(0,j+1,k+\tfrac{1}{2}) - 2*E_z^{n}(0,j,k+\tfrac{1}{2}) \\
+E_z^{n}(0,j-1,k+\tfrac{1}{2}) + E_z^{n}(1,j+1,k+\tfrac{1}{2}) \\
-2*E_z^{n}(1,j,k+\tfrac{1}{2}) + E_z^{n}(1,j-1,k+\tfrac{1}{2}) \\
+E_z^{n}(0,j,k+1\tfrac{1}{2}) - 2*E_z^{n}(0,j,k+\tfrac{1}{2}) \\
+E_z^{n}(0,j,k-\tfrac{1}{2}) + E_z^{n}(1,j,k+1\tfrac{1}{2}) \\
-2*E_z^{n}(1,j,k+\tfrac{1}{2}) + E_z^{n}(1,j,k-\tfrac{1}{2})
\end{bmatrix}
\end{aligned}
\tag{30}
$$

A total of sixteen additions and seven multiplications are required to generate this boundary value. (The leading terms of each multiply turn out to be constant.) Combining terms to decrease the number of floating-point operations gives:

$$
\begin{aligned}
E_z^{n+1}(0,j,k+\tfrac{1}{2}) = & -E_z^{n-1}(1,j,k+\tfrac{1}{2}) \\
& + \frac{c\Delta t-\delta}{c\Delta t+\delta} * \{E_z^{n+1}(1,j,k+\tfrac{1}{2}) + E_z^{n-1}(0,j,k+\tfrac{1}{2})\} \\
& + \frac{2\delta^2-4c\Delta t^2}{c\Delta t+\delta} * \{E_z^{n}(0,j,k+\tfrac{1}{2}) + E_z^{n}(1,j,k+\tfrac{1}{2})\} \\
& + \frac{(c\Delta t)^2}{2\delta(c\Delta t+\delta)} \\
& * \begin{bmatrix}
E_z^{n}(0,j+1,k+\tfrac{1}{2}) + E_z^{n}(0,j-1,k+\tfrac{1}{2}) \\
+E_z^{n}(1,j+1,k+\tfrac{1}{2}) + E_z^{n}(1,j-1,k+\tfrac{1}{2}) \\
+E_z^{n}(0,j,k+1\tfrac{1}{2}) + E_z^{n}(0,j,k-\tfrac{1}{2}) \\
+E_z^{n}(1,j,k+1\tfrac{1}{2}) + E_z^{n}(1,j,k-\tfrac{1}{2})
\end{bmatrix}
\end{aligned}
\tag{31}
$$

The results from this equation are based (in part) on the field values at cells to the left and right, and directly above and below the cell in question.

Simplifying further, the following expression is obtained:

$$\begin{aligned} E_z^{n+1}(0,j,k+\tfrac{1}{2}) = & -E_z^{n-1}(1,j,k+\tfrac{1}{2}) \\ & + K1*\{E_z^{n+1}(1,j,k+\tfrac{1}{2}) + E_z^{n-1}(0,j,k+\tfrac{1}{2})\} \\ & + K2*\{E_z^{n}(0,j,k+\tfrac{1}{2}) + E_z^{n}(1,j,k+\tfrac{1}{2})\} \\ & + K3* \begin{bmatrix} E_z^{n}(0,j+1,k+\tfrac{1}{2}) + E_z^{n}(0,j-1,k+\tfrac{1}{2}) \\ + E_z^{n}(1,j+1,k+\tfrac{1}{2}) + E_z^{n}(1,j-1,k+\tfrac{1}{2}) \\ + E_z^{n}(0,j,k+1\tfrac{1}{2}) + E_z^{n}(0,j,k-\tfrac{1}{2}) \\ + E_z^{n}(1,j,k+1\tfrac{1}{2}) + E_z^{n}(1,j,k-\tfrac{1}{2}) \end{bmatrix} \end{aligned} \tag{32}$$

where

$$K1 = \frac{c\Delta t - \delta}{c\Delta t + \delta} \tag{33}$$

$$K2 = \frac{2\delta^2 - 4c\Delta t^2}{c\Delta t + \delta} \tag{34}$$

$$K3 = \frac{(c\Delta t)^2}{2\delta(c\Delta t + \delta)} \tag{35}$$

This expression now contains only twelve additions and three multiplications. It was decided that this equation could be implemented in hardware as well, so that at the conclusion of this study, the groundwork would be laid for a complete, single board FDTD computational engine capable of generating all cell and boundary field values.

Bibliography

1. Andersh, Dennis J. *Time and Frequency Domain Evaluation of Asymptotic Methods for Computing the Electromagnetic Scattering from Jet Engines*. MS thesis, Air Force Institute of Technology, December 1991.
2. Barkeshli, Kasra and John L. Volakis. "A Vector-Concurrent Application of a Conjugate Gradient FFT Algorithm to Electromagnetic Radiation and Scattering Problems," *IEEE Transactions on Magnetics*, 2892–2894 (July 1989).
3. Bernhardt, Mike, et al. "Paragon Supercomputer/IS-446." Intel Announces the Paragon XP/S Supercomputer; Parallel Architecture Scalable to TeraFLOPS, November 1991.
4. Bernhardt, Mike, et al. "1990 Gordon Bell Prize/IS-436." 1990 Gordon Bell Prize Awarded to Scientists Using Intel iPSC/860 Parallel Supercomputer.
5. Bomans, Luc and Dirk Roose. "Benchmarking the iPSC/2 Hypercube Multiprocessor," *Concurrency* (September 1989).
6. Booch, Grady. *Software Engineering with Ada*. Menlo Park, CA: Benjamin/Cummings Publishing Comp. Inc., 1987.
7. Calalo, R. H., et al. "Hypercube Parallel Architecture Applied to Electromagnetic Scattering Analysis," *IEEE Transactions on Magnetics*, 2898–2900 (July 1989).
8. Carter, Michael B. *Ray Tracing Complex Scenes on a Multiple-Instruction Stream Multiple-Data Stream Concurrent Computer*. MS thesis, Oklahoma State University, 1989.
9. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design*. Reading, MA: Addison - Wesley Publishing Comp., 1989.
10. DeCegama, Angel J. *Parallel Processing Architectures and VLSI Hardware*. Englewood Cliffs NJ: Prentice Hall, 1989.
11. Gustafson, John L. and others. "A Radar Simulation Program for a 1024-Processor Hypercube." *Proceedings of Supercomputing*. 1989.
12. Marek, J. Raley. *An Investigation of a Design for a Finite - Difference Time Domain (FDTD) Accelerator*. MS thesis, Air Force Institute of Technology, December 1991.
13. Marhefka, Ronald J. *Numerical Electromagnetic Code - Basic Scattering Code User's Manual*. Ohio State University Electroscience Laboratory.
14. Patterson, Jean E. and others. "Parallel Computation Applied to Electromagnetic Scattering and Radiation Analysis," *Electomagnetics*, 21–39 (1989).
15. Perlik, Andrew T., et al. "Predicting Scattering of Electromagnetic Fields Using FDTD on a Connection Machine," *IEEE Transactions on Magnetics*, 2910–2912 (July 1989).
16. Reintjes, J. Francis and Godfrey T. Coate. *Principles of Radar*. New York: McGraw Hill Book Company, 1952.

17. Strang, Gilbert. *Linear Algebra and its Applications*. Orlando FL: Harcourt Brace Jovanovich Inc., 1988.

18. Suhr, Scott. *High Frequency Scattering Code in a Distributed Processing Environment*. MS thesis, Air Force Institute of Technology, 1991.

19. Tipler, Paul A. *Physics*. New York: Worth Publishers Inc., 1976.

20. Work, Paul R and Gary B. Lamont. "Efficient Parallelization of Serial Programs for the Intel iPSC/2 and iPSC/860 Hypercubes." *Proceedings of the International Conference of the Intel Supercomputer User's Group*. October 1991.

21. Work, Paul R, et al. "New Computational and Communications Results on the Intel iPSC/860 with the Intel System Software Release 3.3." *Proceedings of the International Conference of the Intel Supercomputer User's Group*. October 1991.

Vita

Paul R Work was born on the 20^{th} of November 1958 in Tyler, Texas. He graduated from Denton Sr High School in Denton, Texas in May 1976 and began attending Brigham Young University in Provo, Utah. He interrupted his studies to serve as a full-time missionary for the Church of Jesus Christ of Latter - Day Saints in Northern Germany in 1978. In 1982, he enlisted in the United States Air Force, and received training in the maintenance and repair of electronic countermeasures equipment and the associated test stations. Stationed initially at RAF Upper Heyford in England, he met and married his wife there. In 1987, he was selected for the Airman Education and Commisioning Program (AECP), and began studying at the University of Missouri-Rolla (UMR) in January 1988. He graduated from UMR summa cum laude in May 1990, and after attending the Air Force Officer Training School in the summer of 1990, he received a commission from the USAF as a second lieutenant on October 1, 1990. His current interests lie in Computer Engineering, dealing with the hardware/software interface, and in performance improvements for existing programs.

Permanent address: 8710 Christygate Lane
Huber Heights, OH 45424